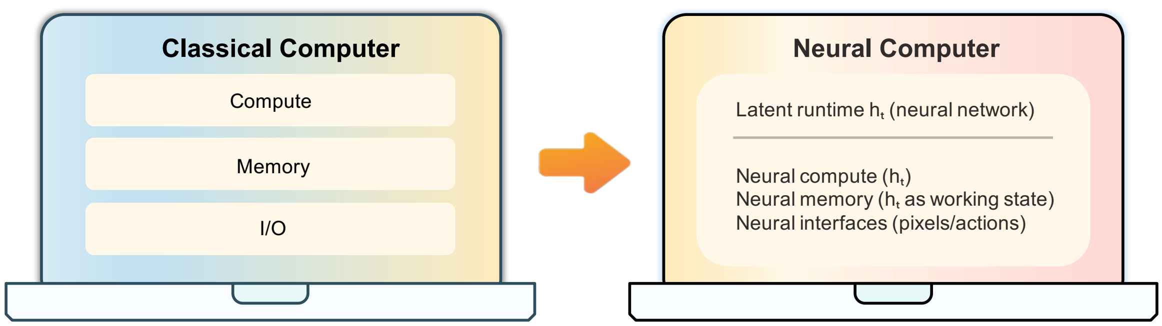
Neural Computers

Mingchen Zhuge[1,2,+], **Changsheng Zhao**[1,+], **Haozhe Liu**[1,2,+], **Zijian Zhou**[1,+], **Shuming Liu**[1,2,+], **Wenyi Wang**[2], **Ernie Chang**[1], **Gael Le Lan**[1], **Junjie Fei**[1,2], **Wenxuan Zhang**[1,2], **Yasheng Sun**[2], **Zhipeng Cai**[1], **Zechun Liu**[1], **Yunyang Xiong**[1], **Yining Yang**[1], **Yuandong Tian**[1], **Yangyang Shi**[1], **Vikas Chandra**[1], **Jürgen Schmidhuber**[2]

[1] Meta AI , [2] KAUST ([+] Core Contributors)

We propose a new frontier: Neural Computers (NCs)—an emerging machine form that unifies computation, memory, and I/O in a learned runtime state. Unlike conventional computers, which execute explicit programs, agents, which act over external execution environments, and world models, which learn environment dynamics, NCs aim to make the model itself the running computer. Our long-term goal is the Completely Neural Computer (CNC): the mature, general-purpose realization of this emerging machine form, with stable execution, explicit reprogramming, and durable capability reuse. As an initial step, we study whether early NC primitives can be learned solely from collected I/O traces, without instrumented program state. Concretely, we instantiate NCs as video models that roll out screen frames from instructions, pixels, and user actions (when available) in CLI and GUI settings. These implementations show that learned runtimes can acquire early interface primitives, especially I/O alignment and short-horizon control, while routine reuse, controlled updates, and symbolic stability remain open. We outline a roadmap toward CNCs around these challenges. If overcome, CNCs could establish a new computing paradigm beyond today's agents, world models, and conventional computers.

Date: April 9, 2026
Correspond: mczhuge@gmail.com, cszhao@meta.com
Blogpost: https://metauto.ai/neuralcomputer



From modular hardware stack to neural latent stack

Contents

1 Introduction

Can a single set of weights act as a "computer"? We term this abstraction a Neural Computer (NC): a neural system that unifies computation, memory, and I/O in a learned runtime state. This usage is distinct from the Neural Turing Machine / Differentiable Neural Computer line (Graves et al., 2014, 2016): our concern is not differentiable external memory, but whether a learning machine can begin to assume the role of the running computer itself.

To implement this idea, we instantiate NCs as video models. At this stage, video models are the most practical substrate for this prototype, though we expect the long-term solution to require a fundamentally new neural architecture (Section 4). This implementation draws on several technical lines. World models (Ha and Schmidhuber, 2018) show that neural networks can internalize environment dynamics and support predictive imagination, while high-capacity video generators such as Veo 3.1 (Google, 2025) and Sora 2 (OpenAI, 2025) show that such learned dynamics can be rendered into coherent frame sequences. Frontier interactive video models such as Genie 3 (Bruce et al., 2024) further extend this trajectory toward action-controllable generative environments. These lines provide practical machinery for current NC prototypes, but do not by themselves define the NC abstraction. In parallel, LLM-driven UI systems such as Imagine with Claude[1] map natural-language inputs to structured interface updates. Yet these capabilities remain split across different systems objects: conventional computers execute explicit programs, agents act through external execution environments, and world models render or predict environment dynamics, while executable state still resides outside the model. NCs are motivated by this gap: they are not a smarter layer on top of the existing stack, but a proposal to make the model itself the running computer. The immediate question in this paper is whether early runtime primitives can be learned directly from raw interface I/O without privileged access to program state.

Throughout this paper, NC denotes this proposed machine form, while CNC denotes its mature, general-purpose realization. We study two interface-specific prototypes of this NC formulation (see Section 2). $\mathrm{NC}_{\mathrm{CLIGen}}$ models terminal interaction from text (natural language or command lines) and an initial frame, while $\mathrm{NC}_{\mathrm{GUIWorld}}$ models desktop interaction from recent pixels and synchronized mouse/keyboard actions (Sections 3.1 and 3.2).

Neural Computer (NC) abstraction (Teaser). A neural system (F, G) parameterized by θ that models an interactive computer interface through a single latent runtime state h_t that carries executable interface state and also acts as working memory (see Eq. (2.1)).

In the $\mathrm{NC}_{\mathrm{CLIGen}}$ 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., fast scrollback, prompt wrapping, window resizing). On carefully scripted data, rollouts can be visually and structurally close to real sessions, and the model can execute short command chains and render their outputs. Arithmetic-probe scores improve substantially with stronger system-level conditioning, though symbolic stability remains limited.

In the $\mathrm{NC}_{\mathrm{GUIWorld}}$ experiments, we evaluate standard world-model designs across action injection, action encoding, and data quality. Figure 1 summarizes this template across two interface-specific NCs trained separately without shared parameters. Qualitatively, the model learns coherent pointer dynamics and short-horizon action responses (e.g., hover/click feedback and window/menu transitions), suggesting that local GUI control primitives are learnable in controlled settings.

Our experimental insights indicate that current NCs already realize early runtime primitives, most notably I/O alignment and short-horizon control. The long-term target is a Completely Neural Computer (CNC), the mature, general-purpose realization of this machine form: a fully learned computer whose compute, memory, and interfaces are unified in a single learned runtime substrate rather than engineered as separate modules. These prototypes are an early step toward that CNC vision. Substantial challenges remain in robust long-horizon reasoning, reliable symbolic processing, stable capability reuse, and explicit runtime governance. Section 4 outlines these open challenges and a roadmap toward CNCs.

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

Completely Neural Computer (CNC) abstraction (Section 4.2). A Neural Computer instance is *complete* (i.e., a CNC) if it is (i) *Turing complete*, (ii) *universally programmable*, (iii) *behavior-consistent* unless explicitly reprogrammed, and (iv) realizes the architectural and programming-language advantages of NCs relative to conventional computers.

Concretely, this work makes the following contributions:

- Define neural computers (NCs) and build video-based prototypes for both CLI and GUI interfaces.
- Provide a data engine and alignment recipe that synchronize text, actions, and frames for the CLI and GUI environments used in this paper.
- Identify practical design choices for NCs through extensive ablation studies.
- Outline an engineering roadmap toward completely neural computers (CNCs), centered on acceptance challenges such as reuse, consistency, and runtime governance.

2 Preliminaries

Throughout this paper, we use *conventional digital computers* as an umbrella term for stored-program machines (e.g., von Neumann-style architectures): at the theory level they are commonly abstracted as random-access machines with an instruction set architecture, and at the systems level they are typically realized through layered operating-system/application stacks. Such systems separate computation, memory, and I/O. Our motivating question is whether a single set of weights can internalize these roles inside one latent runtime state, rather than relying on an external execution environment (e.g., OS/simulator) to carry executable state. We model a video-based neural computer (NC) prototype as a learned latent-state system that folds these roles into an update-and-render loop.

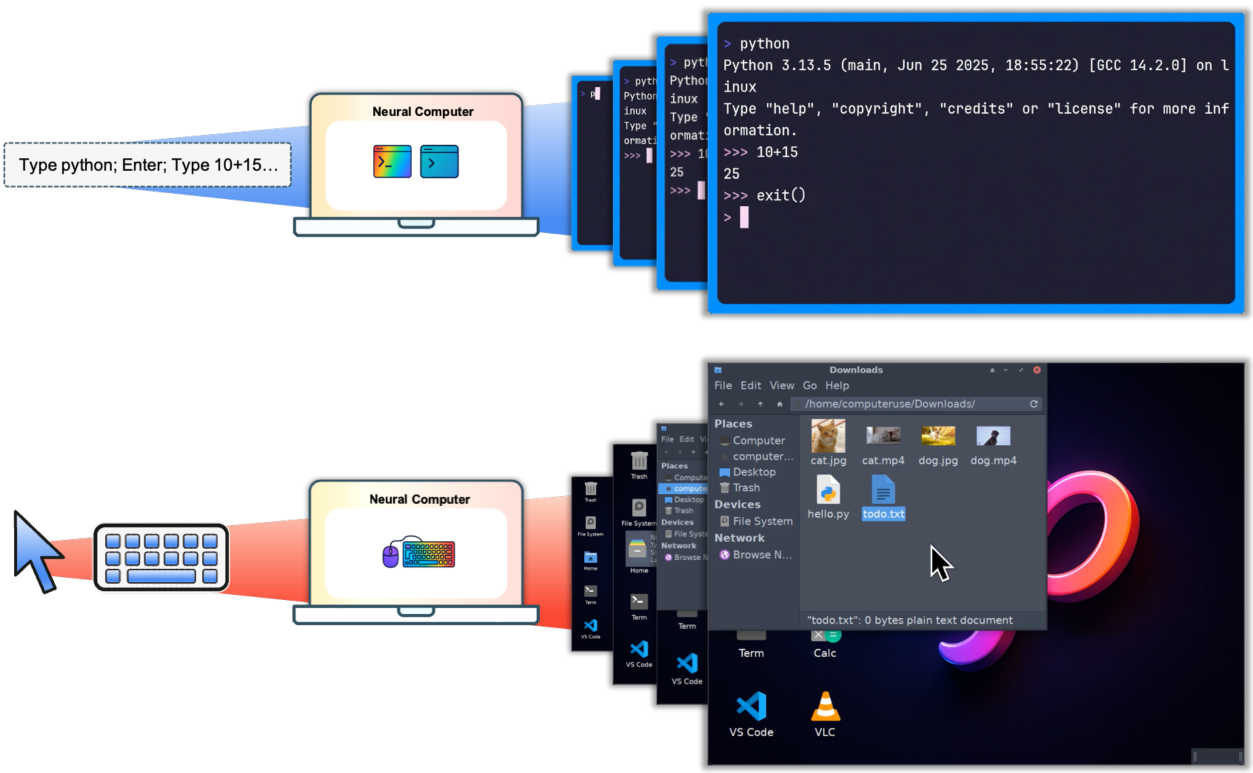


Figure 1 Neural computers across interfaces. Given a prompt or action stream, an NC rolls out future interface frames for / $\mathrm{NC}_{\mathrm{CLIGen}}$ (top) and $\mathrm{NC}_{\mathrm{GUIWorld}}$ (bottom). Logos denote datasets; $\mathrm{NC}_{\mathrm{CLIGen}}$ and $\mathrm{NC}_{\mathrm{GUIWorld}}$ are the corresponding models trained on those datasets. It models terminal or desktop dynamics.

Specifically, an NC updates a latent runtime state from the current observation and conditioning input, and then predicts (or samples) the next observation. In this paper, we treat screen frames as observables and define actions as time-indexed conditions. 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 runtime state and samples the next frame x_{t+1}. Formally, an NC defined by an initial runtime state h_0, an update function F_θ, and a decoder G_θ operates as follows, where G_θ parameterizes a distribution over next frames:

$$h_t = F_\theta(h_{t-1}, x_t, u_t), \qquad x_{t+1} \sim G_\theta(h_t). \tag{2.1}$$

In this formulation, h_t provides the persistent runtime memory, F_θ carries the state-update computation, and (x_t, u_t, G_θ) define the I/O pathway from observations and actions to the next observable state.

Notation. We use h_t for the NC latent runtime state and reserve z for VAE/video latents used in diffusion-style video models (e.g., Section 3.2).

This update-and-render loop can be described using world-model terminology, where x_t are observations and u_t provides conditioning. In that terminology, the input sequence $\{u_t\}$ is referred to as a conditioning stream. This view supplies practical machinery for the current prototype, but an NC is not merely a predictor of interface dynamics: it is a learned runtime mechanism in which the latent state h_t carries executable context, F_θ integrates new observations and inputs, and G_θ renders the next frame. Auxiliary heads can encode and decode prompts, buffers, or action traces, shifting functionality that would traditionally live in OS queues, device drivers, and UI toolkits into latent-state dynamics.

2.1 Related Work

Early neuromorphic designs (Mead and Ismail, 2012) explored neural computation as a physical substrate. Differentiable memory and program-execution architectures, including fast weight programmers (Schmidhuber, 1992, 1993b,a), Neural Turing Machines (Graves et al., 2014), Differentiable Neural Computers (Graves et al., 2016), and Neural Programmer-Interpreters (Reed and De Freitas, 2015), showed that neural controllers with memory can execute structured procedures. Differentiable world models (Schmidhuber, 1990, 2015) learn neural representations of environment dynamics, and inspire our update-and-render formulation. Latent video and world models (Ha and Schmidhuber, 2018; Hafner et al., 2019b,a; Bruce et al., 2024) apply these ideas to embodied control and interactive environments. Genie 3 (Bruce et al., 2024), in particular, frames such models as agent-training substrates with improved physical consistency. More recently, high-capacity generators such as Veo 3 (Google, 2025) and Sora 2 (OpenAI, 2025) emphasize open-ended, photorealistic simulation. In parallel, systems such as NeuralOS (Rivard et al., 2025) and Imagine with Claude (Anthropic, 2025) bring model-based conditioning to desktop and DOM-style interfaces. Building on this trajectory, we study two NC instantiations for CLI and GUI with interface-specific conditioning, supported by a shared data engine and a staged roadmap toward the CNC vision.

3 Implementation of Neural Computers

We build on the Wan2.1 model (Wan et al., 2025), which was a state-of-the-art video generation model at the time of our experiments. We add NC-specific conditioning and action modules, together with interface-specific training recipes. Figure 1 illustrates this setup: NCs take a prompt or action stream as input and generate future interface frames in both CLI and GUI settings. In the present prototypes, these prompts and actions are logged conditioning streams, so evaluation remains open-loop rather than closed-loop interaction with a live environment. We refer to these two instantiations as CLIGen (CLI; Section 3.1) and GUIWorld (GUI; Section 3.2).

In this video-based instantiation, the NC latent runtime state h_t is realized by the model's time-indexed video latents z_t. Under this abstraction, the diffusion transformer acts as the state-update map: it consumes prior latents together with the current observation and conditioning inputs, and produces the updated state h_t (realized as z_t). The decoder G_θ parameterizes a distribution over the next frame x_{t+1}. Auxiliary heads encode and decode conditioning streams u_t, including text prompts and action traces. Structured logs such as terminal buffers are used for alignment and evaluation where available, not as privileged model-state inputs.

Figure 2 Data types used to learn NC behaviors. Logos denote datasets: CLIGen (General) replays public `asciinema` traces spanning diverse real-world terminal workflows. CLIGen (Clean) uses scripted `vhs` runs to capture deterministic terminal traces for controlled experiments. GUIWorld captures desktop RGB with synchronized mouse/keyboard traces to validate action-conditioned rendering and control on GUIs.

3.1 / The CLI Video Generators

CLIGen instantiates the NC abstraction in command-line interfaces. 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 video latent state z_t implements the latent runtime state h_t by tracking 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). We use two CLI datasets: CLIGen (General), which contains diverse, open-ended terminal traces, and CLIGen (Clean), which contains deterministic Dockerized traces. We train one $\mathrm{NC}_{\mathrm{CLIGen}}$ model per dataset under the same architecture.

Dataset	**Data characteristics**	**Model**
CLIGen (General)	diverse, open-ended terminal traces	$\mathrm{NC}_{\mathrm{CLIGen}}$
CLIGen (Clean)	deterministic Dockerized scripts with cleaner, better-paced buffers	$\mathrm{NC}_{\mathrm{CLIGen}}$

3.1.1 Data pipeline

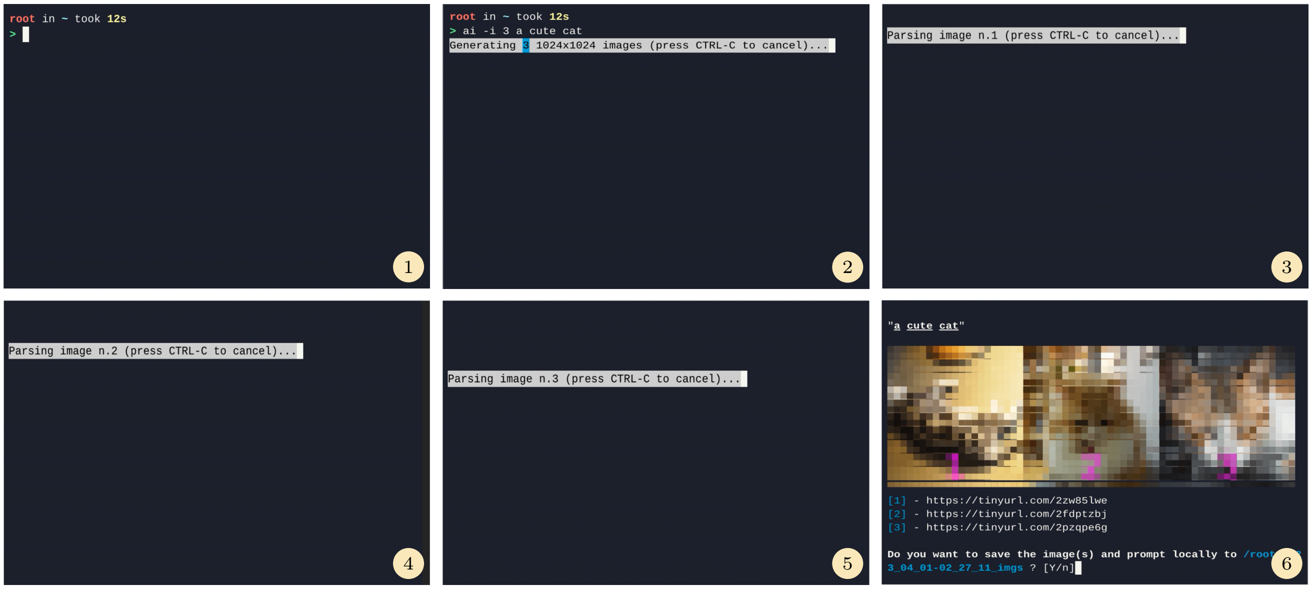
The CLIGen (General) dataset is built from publicly available `asciinema` `.cast` trajectories[2]. The `asciinema` stack records and replays terminal sessions with synchronized timing and ANSI-faithful decoding. We replay each session with the official tools and render it into terminal frames, preserving 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 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

[2] https://asciinema.org/

Table 1 Data 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`.

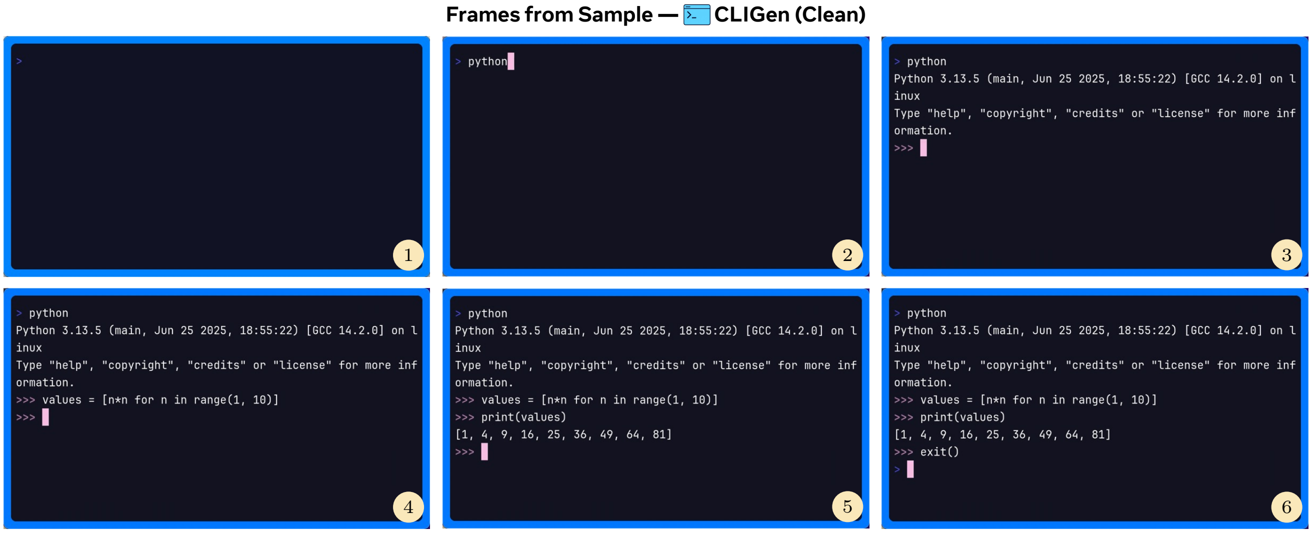
Regular

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.

styles (semantic, regular, and detailed), which serve as prompts. As shown in Figure 2 (left), this split spans diverse real-world terminal use cases[3].

The CLIGen (Clean) dataset is collected using the open-source `vhs` 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 final frame to normalize length[4].

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.

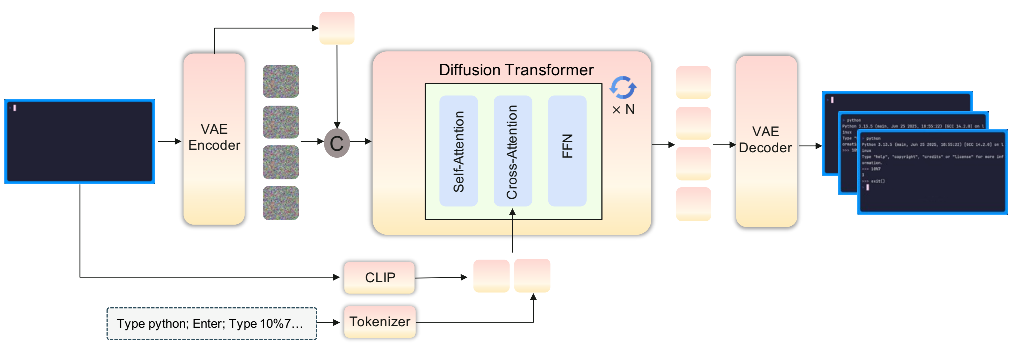


Figure 3 / $\mathrm{NC}_{\mathrm{CLIGen}}$ architecture. Terminal frames are observations x_t. A prompt and the first frame seed the conditioning stream. The Wan2.1-based latent state z_t rolls forward under the standard I2V sampling scheme.

3.1.3 Implementation Details

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 ∼15,000 H100 GPU hours at batch size 1. Training on CLIGen (Clean) across both subsets requires ∼7,000 H100 GPU hours.

3.1.4 Evaluations

Unless otherwise noted, NC in this section refers to the current video-based CLI prototype. We report six practical takeaways:

[3]Additional preprocessing details and a .cast example are in Sections B and C.1, with a sample overview in Table 1.
[4]Additional details are provided in Sections B and C.2, with a representative data sample in Table 1.

① The NC maintains high-fidelity terminal rendering at practical font sizes (e.g., 13 px), preserving readable interface state.

② Prompt specificity is an effective control channel: detailed, literal captions improve text-to-pixel alignment.

③ On clean but domain-specific data, global PSNR/SSIM plateau around 25k steps (Figure 5), indicating early saturation in reconstruction metrics rather than a complete halt in learning.

④ The NC reproduces complex terminal appearances while sustaining coherent short-horizon command rollouts under fixed conditioning.

⑤ Symbolic computation remains the main bottleneck: structured arithmetic reveals reliability limits, motivating stronger symbolic or system-level conditioning.

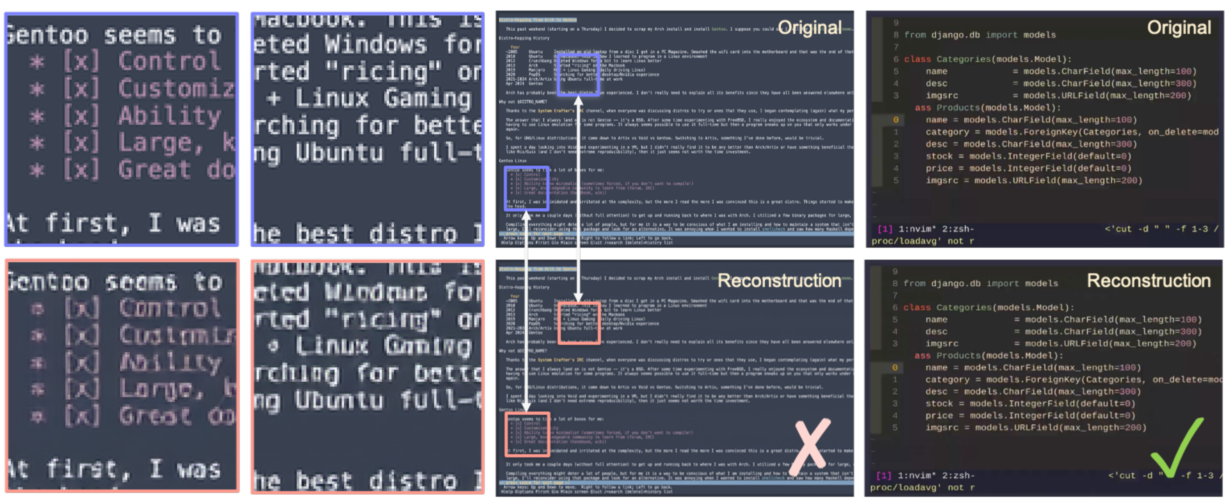
⑥ In our setting, without changing the NC backbone or adding RL, reprompting improves symbolic probes (4%→83%; Figure 6), reinforcing the view that current models are strong renderers and conditionable interfaces rather than native reasoners (Table 6).

Experiment 1: The NC stays readable at practical font sizes

Table 2 Reconstruction quality.

Metric	Value
Average PSNR	40.77
Average SSIM	0.989

Concurrent work (Rivard et al., 2025) argues that generic natural-image VAEs can perform poorly on structured computer screenshots. We test 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, it is high (40.77 dB PSNR, 0.989 SSIM). At 6 px, text exhibits noticeable blurring even when global PSNR/SSIM remain strong, because background regions dominate these metrics.



(a) dense, 6px (b) regular, 13px

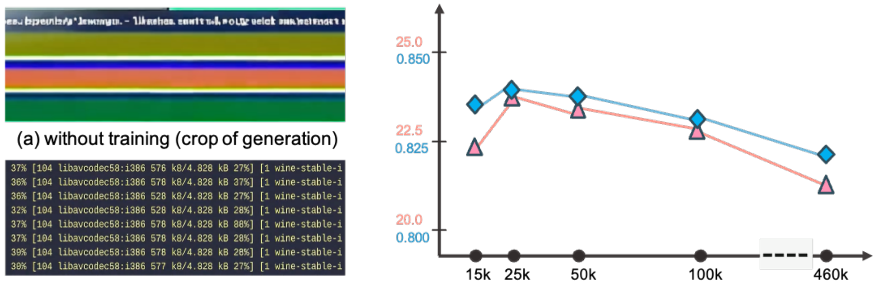
Figure 4 Wan2.1 VAE reconstructions on CLIGen (General) terminal frames at different font sizes.

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 NC training.

Experiment 2: Performance plateaus early and can degrade with prolonged training

On clean but domain-specific structured interfaces, global reconstruction metrics improve rapidly early and then show limited additional gains under the current training objective. In CLIGen (Clean), PSNR/SSIM plateau quickly, suggesting that further optimization becomes bottlenecked less by model capacity than by the quality and pacing of the available supervision. After the early gains, the remaining errors are often tied to artifact-prone signals (e.g., rendering glitches or rapid screen changes that disrupt temporal alignment), so additional training on the same objective can yield diminishing or even slightly unstable returns in these perceptual metrics.

Panels (a–b) illustrate the effect of training on CLIGen data. Without CLIGen fine-tuning, Wan2.1 produces garbled terminal outputs (a). After 25k steps, the model generates readable text with consistent formatting and color cues (b).



(a) without training (crop of generation)

(b) at 25k steps (crop of generation)

(c) Inference scores at different training steps

Figure 5 (a–b) Qualitative generations before and after CLIGen training; (c) CLIGen (Clean) PSNR/SSIM plateau around 25k training steps.

Figure 5 plots the corresponding PSNR/SSIM curves and shows that these global perceptual metrics flatten around 25k steps. They improve little with further training up to 460k steps, and extended optimization can even slightly reduce them. One plausible explanation is that most learnable structured patterns are acquired early, and further gains require higher-quality, better-paced, or more informative supervision.

Experiment 3: Literal captions drive rendering accuracy

Caption specificity has a strong effect on terminal rendering quality. As shown in Table 3, detailed, literal descriptions improve reconstruction fidelity. PSNRincreases from 21.90 dB (semantic) to 26.89 dB (detailed), a gain of nearly 5 dB, compared to less specific, high-level semantic descriptions.

The three caption tiers correspond to the same underlying terminal sequence but differ in length and granularity. **Semantic** captions (average 55 words) provide high-level summaries (e.g., "a terminal session generates three cat images"). **Regular** captions (average 52 words) include key commands and outputs (e.g., `ai -i 3 a cute cat`, status messages). **Detailed** captions (average 76 words) transcribe screen content more exhaustively, including exact text, colors, and formatting.

This progression helps explain why literal descriptions are particularly effective for terminal rendering. Unlike natural images, which are dominated by global style patterns, terminal frames are governed primarily by text placement. Detailed captions act as scaffolding—explicitly specifying which tokens appear where—thereby enabling precise text-to-pixel alignment.

Table 3 Caption styles versus TI2V 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**

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

Beyond PSNR and SSIM, character-level accuracy is a more direct metric for terminal rendering. Character-level accuracy requires explicit pixel-to-text correspondence. For CLIGen (Clean), we apply Tesseract to five uniformly sampled (ground-truth, generated) frame pairs per video and normalize whitespace. We then compute two metrics (full protocol in Appendix B). Character accuracy uses the Levenshtein distance between concatenated ground-truth and generated texts. Exact-line accuracy measures the fraction of ground-truth lines whose normalized content exactly matches the prediction at the same line index.

Table 4 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 4 shows that our models achieve substantial text rendering accuracy under this protocol. Character accuracy increases from 0.03 at initialization to 0.54 at 60k steps, with exact-line matches reaching 0.31 (0.26 by 40k). Most gains occur within the first 40k steps, followed by smaller refinements thereafter. These OCR-based metrics capture properties beyond perceptual similarity. Accurately generating terminal characters requires modeling text structure, font rendering, and spatial relationships. These are core competencies for interactive neural computer systems. This level of character-level precision is a step toward usable, not just plausible, terminal interfaces. At the same time, we interpret this result primarily as evidence of interface fidelity, while routine reuse and native symbolic computation remain separate questions.

Experiment 5: Does this NC instantiation show native CLI reasoning?

We also probe symbolic computation with CLI arithmetic tasks. These tasks are a sharp stress test for symbolic reliability: humans answer them instantly, yet current NC instantiations often fail on seemingly simple symbolic operations.

Table 5 Arithmetic probe accuracy (100 problems sampled from a 1,000-problem held-out pool).

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

Our arithmetic probe presents basic mathematical operations through terminal interactions. We reserve a held-out pool of 1,000 math problems and randomly sample 100 problems as the final evaluation set. Table 5 shows that current video models, including this NC instantiation, struggle on these symbolic 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 on these fundamental tasks. These results contrast with common claims of strong symbolic reasoning in current video models. Sora2's 71% accuracy is a notable outlier and may reflect system-level advantages or additional training beyond our current setup. Overall, native symbolic reasoning remains an open challenge for current video-based NC instantiations. Accordingly, arithmetic probes in this paper serve as a targeted test of symbolic stability under the current prototype substrate.

The poor arithmetic-probe performance in Table 5 raises a key question. Does this prototype require specialized reinforcement learning to achieve reliable symbolic computation, or can stronger conditioning substantially narrow this gap?

Experiment 6: Does this NC instantiation require RL for symbolic probes?

As shown in Figure 6, $\mathrm{NC}_{\mathrm{CLIGen}}$ accuracy on CLIGen (Clean) arithmetic tasks rises from 4% to 83% under reprompting. This suggests that system-level conditioning can be an effective first lever for improving performance on symbolic probes. It is complementary to (rather than strictly requiring) RL-based training pipelines. More generally, the success of reprompting highlights how sensitive symbolic-probe outcomes are to the conditioning interface. Much of the apparent "reasoning" gain can come from better specification and instruction-following rather than new native computation. For the arithmetic subset, we include the correct answer explicitly in roughly half of the training captions to encourage reliable rendering of the output string. Because reprompting can similarly provide stronger hints (or even outsource computation to an external text system), we interpret the gain primarily as evidence of steerability. It also shows faithful rendering of conditioned symbolic content. We do not treat it as a clean demonstration that the NC backbone performs arithmetic internally.

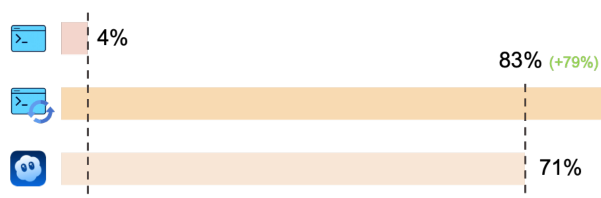


Figure 6 Reprompting boosts performance to 83%.

Table 6 Hypotheses for Sora2's advantage.

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

The evidence supports system-level conditioning as a practical path forward for this NC instantiation. Among the three hypotheses for improving arithmetic-probe performance—stronger base models, reinforcement learning, or enhanced conditioning—our results most strongly favor the third approach. The gain from reprompting (4%→83%), achieved without modifying the underlying NC backbone, is substantial. It shows that measured "reasoning" on these probes is highly sensitive to specification and conditioning. We therefore do not treat it as direct evidence of native arithmetic inside the NC backbone.

In our setting, strategic conditioning yields larger symbolic-probe gains than the RL pipeline we tested. Evaluations should therefore distinguish native computation from conditioning-assisted performance when assessing reasoning capabilities in current video-based NC instantiations.

3.1.5 Visualizations

(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, classic shell prompts and system outputs, long-running jobs with rapidly scrolling and color-coded package logs, full-screen TUIs such as partition editors, 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.

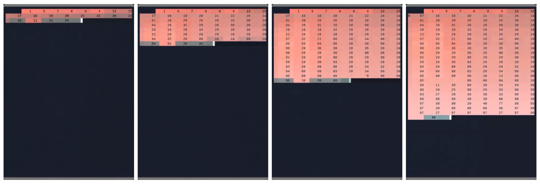
(2) CLIGen (Clean) REPL visualizations. In contrast to open-world traces, CLIGen (Clean) REPL samples are scripted and temporally well-paced (Figures 16–19; additional examples are in Appendix C). Each sample includes an explicit action trace (e.g., `Sleep`, `Type`, `Enter`, arrow keys, `Hide`) alongside rendered terminal frames, making the action-to-pixel link visually unambiguous. The key insight is that these scripted traces isolate rendering-and-control errors from semantic ambiguity: with explicit actions, failures are dominated by low-level mechanics (cursor placement, character edits, monospace alignment, line breaks, temporal consistency).

(3) CLIGen (Clean) math visualizations. Figures 20–22 compare math REPL rollouts, and Figures 23–25 show reprompting cases. Together they highlight why arithmetic probes should separate native computation from answer-conditioned rendering. All full-resolution pages are in Appendix E; below we keep clickable thumbnails at the original location for quick navigation.

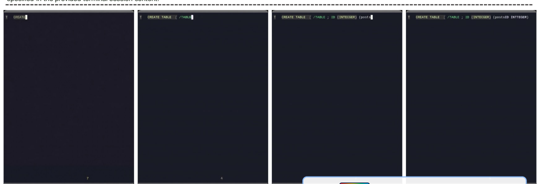
CLIGen Visualization Thumbnails

Click any thumbnail to jump to its full-resolution page in Appendix

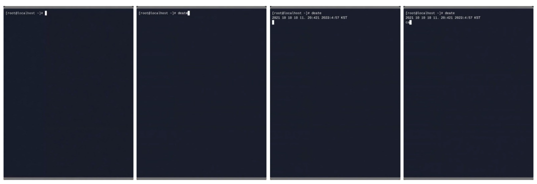
CLIGen (General) Visualizations



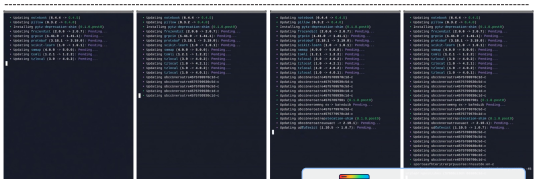
The terminal displays a series of ANSI escape code formatted texts with changing background and foreground colors, executing commands like `\\u001b[48;2;255;128;128;38;2;0;0;0m` which set the background to a shade of pink and text to black, and printing numbered lists with colors. The output includes 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.



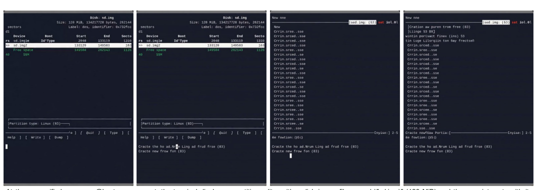
The user types the command `CREATE TABLE posts (ID INTEGER)`, with the terminal display ... including green and yellow text, and the cursor moving character-by-character as the user types, ... the command being executed, with key words like `CREATE` and `TABLE` in distinct colors, and ...



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`, an... and installing statuses, including percentages, for packages like `smmap`, `tomli`, and `protobuf`, ...



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\".

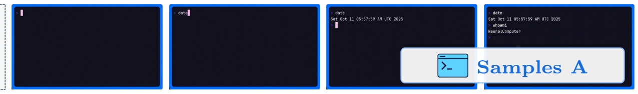
The terminal displays a progress bar with the command output \"Evaluating\" and percentages fr... such as \"|█████████████████| \" to \"|█████████████████████| \" ... to \"0:00:07\". The output includes specific item completion and estimated time remaining, with the ...

CLIGen (Clean) Visualizations

Sleep 500ms
Type "echo $HOME"
Sleep 90ms
Enter
Sleep 1442ms
Hide

Sleep 200ms
Type "env | head -n 5"
Enter
Sleep 600ms
Hide

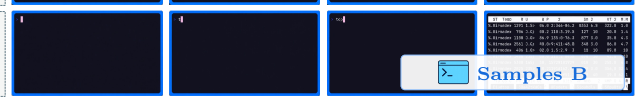
Sleep 200ms
Type "date"
Enter
Sleep 300ms
Type "whoami"
Enter
Sleep 300ms



Sleep 500ms
Type "echo $PATH"
Sleep 90ms
Enter
Sleep 1442ms
Hide

Sleep 400ms
Type "seq 1 28 | paste -d',' - - - - | column -t -s',' | tee metrics_7x4.txt"
Enter
Sleep 2000ms
Hide

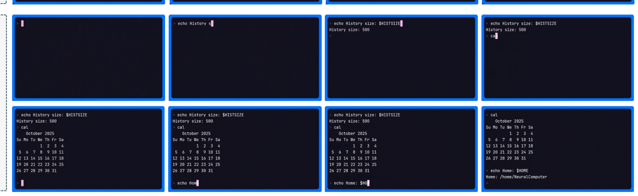
Sleep 200ms
Type "top"
Enter
Sleep 2s
Down 3
Sleep 600ms
Up 2
Hide



Sleep 400ms
Type "echo Home: $HOME"
Enter
Sleep 600ms
Type "date +%H:%M:%S"
Enter
Sleep 600ms
Type "printenv HOME"
Enter
Sleep 600ms
Sleep 400ms
Hide

Sleep 400ms
Type "python - <<'PY'"
Enter
Type "import time"
Enter
Type "for i in range(18):"
Enter
Type "   print(f'Frame {i:02d} ::' + '>' * (i % 20))"
Enter
Type "   time.sleep(0.2)"
Enter
Type "PY"
Enter
Sleep 4000ms
Hide

Sleep 180ms
Type "echo History size: $HISTSIZE"
Sleep 120ms
Enter
Sleep 400ms
Type "cal"
Sleep 120ms
Enter
Sleep 400ms
Type "echo Home: $HOME"
Sleep 120ms
Enter
Sleep 400ms
Sleep 400ms
Hide



Sleep 800ms
Sleep 180ms
Type "echo Learning shell basics"
Sleep 120ms
Enter
Sleep 400ms
Type "date +%Y-%m-%d"
Sleep 120ms
Enter
Sleep 400ms
Type "echo Login shell: $0"
Sleep 120ms
Enter
Sleep 400ms
Type "uname -r"
Sleep 120ms
Enter
Sleep 400ms
Sleep 400ms
Hide

CLIGen Visualization Thumbnails

Click any thumbnail to jump to its full-resolution page in Appendix

CLIGen (Clean) Math Comparison

CLIGen (Clean) Math Reprompting

3.2 The GUI World Models

We also instantiate the NC abstraction in interactive desktop environments with $\mathrm{NC}_{\mathrm{GUIWorld}}$. In this setting, fine-grained action control is essential: GUI interaction requires precise cursor tracking, timely click feedback, and robustness to rapidly changing interface states. We model each interaction as a synchronized sequence of RGB frames x_t and input events u_t (mouse and keyboard). The latent video state maintains interface context across frames, while temporally aligned action inputs provide control signals designed to preserve pixel-level correspondence between user actions and visual changes.

3.2.1 Data pipeline

Table 7 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

The dataset includes two styles of random interaction: "Random Slow" and "Random Fast", plus a smaller set of supervised trajectories collected by Claude CUA (Anthropic). 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. The supervised trajectories are approximately 110 hours. These goal-directed traces provide higher-signal action–response pairs without overwhelming the exploration data. Table 7 summarizes cursor and action statistics across splits; in the collected CUA trajectories, action density is lower due to latency introduced by Claude's tool API between successive steps.

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. In dataset packaging, we store both raw-action and meta-action views for modeling. This lets us train either the raw-action or meta-action encoder under the same loss stack[5].

3.2.2 Model architecture

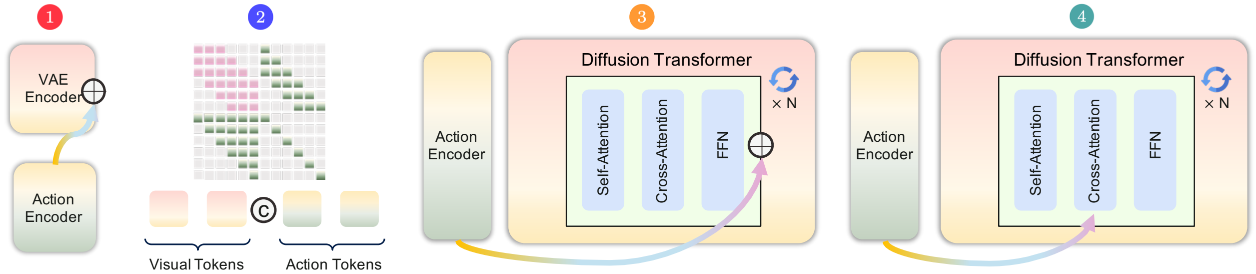
The GUIWorld architecture builds on the Wan2.1 (Wan et al., 2025) by incorporating explicit action-conditioning modules. The central challenge is to align time-stamped user actions with generated frames and inject this information at the appropriate depth within the transformer.

Action features are encoded on-the-fly from frame-aligned mouse and keyboard signals (Section 3.2.1). 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 differences in rendering fidelity and control behavior are modest[6].

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 11.

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

[5] Conversion details and alignment quality appear in Appendix D.
[6] Appendix Table 17 summarizes the representational differences.



Mode	Injection point	Notes
1 `external`	Conditioned VAE input	Actions are injected as an external conditioning stream that 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 attention mask, enabling mid-level fusion.
3 `residual`	Hidden state with action delta	Actions are added through residual-style modulation branches, acting as a strong but more indirect baseline.
4 `internal`	After cross-attn, before FFN	Actions enter through dedicated cross-attention inside the transformer blocks, fusing near the backbone core.

Figure 7 Four modes for injecting GUI actions into the diffusion transformer. 1 modulates VAE latents before the transformer; 2 adds action tokens alongside visual tokens; 3 applies block-wise residual updates; and 4 inserts an action cross-attention module inside transformer blocks.

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 $z'_{1:T}$.

Formally, given VAE latents $z_{1:T}$ and temporally aligned action features $u_{1:T}$, an external action module applies a small stack of temporal self-attention and action cross-attention layers. This produces a residual update $\Delta z_{1:T}(u_{1:T})$. The modified latents are

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

and the diffusion transformer operates solely on $z'_{1:T}$. The diffusion backbone remains unchanged, and action features are not exposed as explicit tokens within the transformer.

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 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 12). 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 an external residual branch. This follows the `residual` conditioning paradigm introduced by 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 $\Delta h^{(l)}(a, \text{mouse})$. The updated hidden states are given by

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

which are passed to the subsequent transformer block. 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 directly 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

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

where SA denotes self-attention and $\mathrm{CA}_{\text{text}}$ and $\mathrm{CA}_{\text{action}}$ denote the text and action cross-attention modules applied in sequence. 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 yields the best SSIM/FVD trade-off for fine-grained GUI interaction in our ablations. 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 Implementation Details

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. 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 D). Runs use 64 GPUs for about 15 days, totaling about 23k GPU-hours per full pass.

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.

3.2.4 Evaluation setup

Our ablations target three capabilities: global fidelity, post-action responsiveness, and cursor-control precision. We use the FVD/LPIPS/SSIM suite as the core metrics. We also add 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 post-action clips. Ablations vary conditioning design and action encoding to measure how these choices affect perceptual quality and responsiveness when rolled out against ground-truth interfaces[7].

[7] Full metric definitions and implementation details are provided in Appendix B.3.

1. In GUIWorld, a small amount of goal-directed data outperforms much larger random exploration, showing that alignment quality matters more than nominal scale for action–response learning.
2. Precise cursor control requires explicit visual supervision: SVG mask/reference conditioning raises cursor accuracy to 98.7%, indicating that local GUI control primitives are learnable in controlled settings.
3. Action injection depth matters: relative to shallow `external` conditioning, `contextual`, `residual`, and especially `internal` fusion improve post-action responsiveness and visual consistency.
4. Action representation also matters: under the same injection mode, API-like meta-actions consistently outperform raw event-stream encoding.

Experiment 7: Data quality dominates performance

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 8). However, high-frequency cursor jitter and irregular, non-goal-directed action bursts make consistent control difficult under dense, stochastic input streams.

Table 8 Overall performance across data sources.

Split	FVD_{all}	$SSIM_{all}$	$LPIPS_{all}$
Untrained baseline	149.61	0.496	0.605
Random Fast (train)	48.17	0.695	0.483
Random Slow (train)	20.37	0.830	0.237
Claude CUA (train)	14.72	0.885	0.144

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. At the current stage, this result primarily indicates that alignment quality matters more than nominal scale for learning action–response structure in NC prototypes.

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. A natural baseline is to condition on normalized cursor-coordinate sequences `mouse_trajectories` $\subset [0,1]^{T\times 2}$ (details in Appendix D.4). To strengthen this signal, we further encode the normalized trajectories using a Fourier mouse encoder. We map coordinates to $[-1,1]^2$ and project them through a fixed Gaussian matrix to obtain random Fourier features. A small MLP produces

Table 9 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%**

per-frame embeddings, which we aggregate into lag-aware windows aligned with the VAE stride. The resulting latent action sequence conditions the action modules and participates in the temporal contrastive loss.

However, Table 9 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 importance of precise cursor placement, we introduce explicit visual cursor supervision. We render an SVG cursor at each (x_t, y_t) to produce per-frame cursor masks m_t and cursor-only foregrounds f_t (right panel of Figure 8). Following Figure 8, we construct a reference stream. The first frame contains the full desktop image, while subsequent frames contain only the cursor foreground over a neutral background, masked to the cursor region. We encode both the video and reference streams with the shared VAE, yielding video latents $z_{1:T}$, reference latents $z_{1:T}^{\mathrm{ref}}$, and mask tags $\tau_{1:T}$. The diffusion transformer receives the concatenated tensor

$$\mathrm{concat}\big(z_{1:T},\ \tau_{1:T},\ z_{1:T}^{\mathrm{ref}}\big),$$

which anchors the static GUI layout at $t{=}0$ and provides localized supervision around the cursor for $t{>}0$.

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. At the same time, this result is best viewed as evidence that local GUI control primitives are learnable under explicit supervision in controlled settings.

Experiment 9: Action injection under different schemes

Table 10 Action-driven metrics across injection schemes (15 frames after action).

Mode	SSIM$_{+15}$ ↑	LPIPS$_{+15}$ ↓	FVD$_{+15}$ ↓
`baseline`$_1$ (untrained)	0.326	0.649	184.3
`baseline`$_2$† (external)	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**

† `baseline`$_2$ (`external`) was early-stopped at ∼50% of the planned training budget after preliminary rollouts did not warrant further compute. Included only as a rough reference.

Holding data and the action encoder fixed, we compare injection schemes on clean runs (Table 10). We compute action-driven metrics over the 15 frames following each click, scroll, or key event. Relative to both baselines (untrained and `external`), mid- and deep-level fusion yields consistent improvements in post-action quality. This includes `contextual`, `residual`, and `internal` injection.

Specifically, moving from input-level conditioning (`external`) to token-level fusion (`contextual`) improves SSIM from 0.746 to 0.813 and reduces FVD from 33.4 to 24.8. Deeper injection sharpens these gains. `internal` achieves the highest structural consistency (SSIM 0.863) and the lowest temporal distortion (FVD 14.5), while `residual` attains the lowest perceptual distance (LPIPS 0.138). Together, these trends associate deeper action injection with improved tracking of fine-grained cursor motion and layout changes. [8]

Experiment 10: Do action encodings matter?

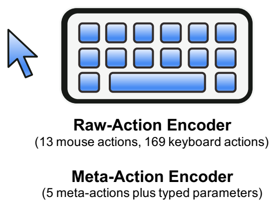
Table 11 Raw-action vs. API-like action encoding under the same injection mode (15 frames after action).

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

We compare two action encodings under the same injection mode to isolate the effect of representation choice (Table 11). Under `internal` conditioning, the meta-action (API-like) encoding yields small but consistent

[8] Appendix D summarizes the corresponding injection schemes and alignment details.

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

improvements over the raw-action representation. SSIM increases from 0.847 to 0.863, LPIPS drops from 0.144 to 0.141, and FVD drops from 16.6 to 14.5. However, these gains are modest compared to the substantially larger improvements observed when varying the action injection scheme itself (Table 10). This suggests that encoding granularity is not the dominant factor governing GUI interaction fidelity.

Table 12 contrasts how short commands and shortcuts (e.g., `ls -l`, `ctrl+v`) are represented under the two encodings. 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.

3.2.5 Visualizations

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 8; Figures 26–30). Consistent with the action-driven metrics in Table 10, 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 31–33 emphasize how small low-level deviations compound. Figures 34–36 focus on numeric/UI fidelity and interaction semantics. Figures 37–39 add stress cases where correctness hinges on precise field edits and page state. All full-resolution pages are in Appendix E; below we keep clickable thumbnails at the original location for quick navigation.

GUIWorld Visualization Thumbnails

Click any thumbnail to jump to its full-resolution page in Appendix

GUIWorld Samples 1–5

Claude
Collect Ground Truth Video with CUA

"thinking": "Now I'll click on the Term icon to open the terminal:"
"action": "left_click", "x": 82, "y" : 520

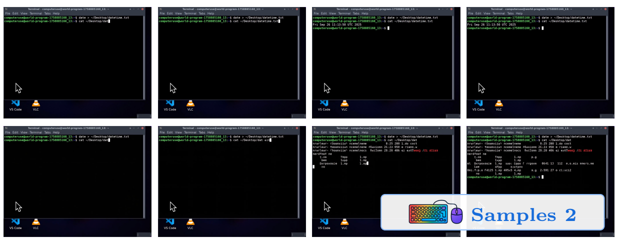
Ground Truth

Generation
First frame and actions as input

Claude
Collect Ground Truth Video with CUA

"thinking": "Now I'll press Enter to execute the command:"
"action": "type", "text": "cat ~/Desktop/datetime.txt"

Ground Truth

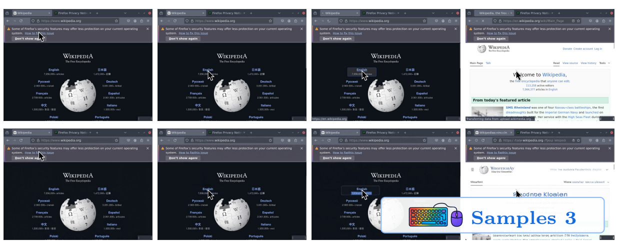


Generation
First frame and actions as input

Claude
Collect Ground Truth Video with CUA

"thinking": "I see Wikipedia appears in the suggestions. I'll click on it to go to Wikipedia:"
"action": "left_click", "x": 240, "y" : 150

Ground Truth

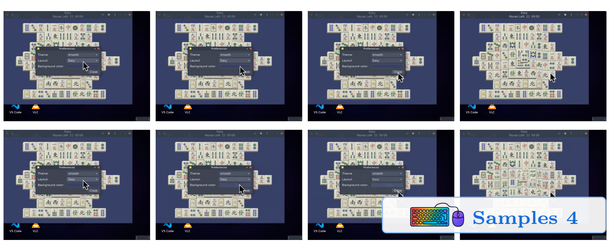


Generation
First frame and actions as input

Claude
Collect Ground Truth Video with CUA

"thinking": "I can see that the theme has changed to \"smooth\" and I can already see the tiles in the background have a different style. Now I need to click the \"Close\" button to apply these changes and close the preferences dialog:"
"action": "left_click", "x": 629, "y" : 423

Ground Truth

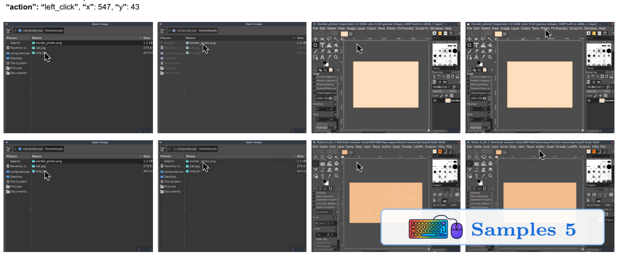


Generation
First frame and actions as input

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

Ground Truth



Generation
First frame and actions as input

GUIWorld Samples 6–10

Claude
Collect Ground Truth Video with CUA

"thinking": "I see I've clicked the \"2\" button. Now I need to click the \"0\" button twice to make \"200\": "
"action": "left_click", "x": 43, "y" : 465

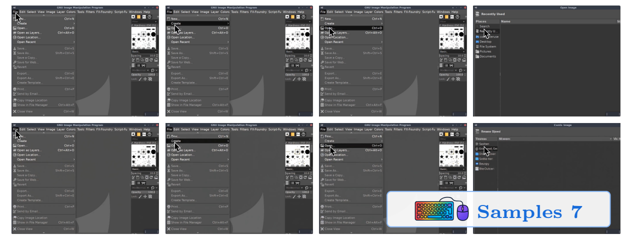
Ground Truth

Generation
First frame and actions as input

Claude
Collect Ground Truth Video with CUA

"thinking": "Perfect! GIMP is now open. Let's import an image from the Downloads folder. I'll click on the File menu and select Open:"
"action": "left_click", "x": 43, "y" : 465

Ground Truth

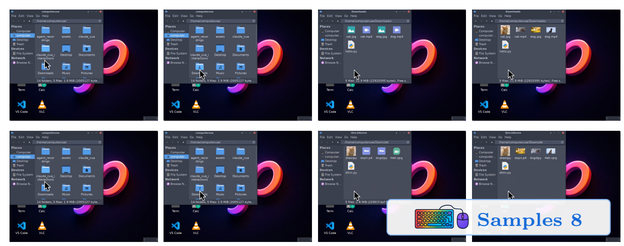


Generation
First frame and actions as input

Claude
Collect Ground Truth Video with CUA

"thinking": "Excellent! Nautilus is open and showing the home directory. Now I need to navigate to the Downloads folder. I can see the Downloads folder in the current view, so I'll double-click on it:"
"action": "double_click", "x": 248, "y" : 408

Ground Truth

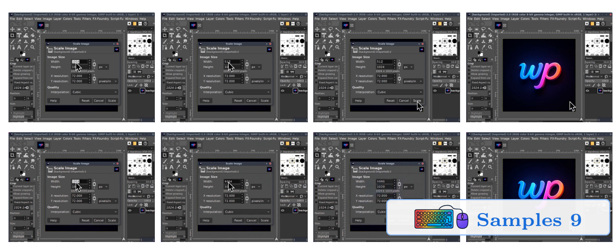


Generation
First frame and actions as input

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

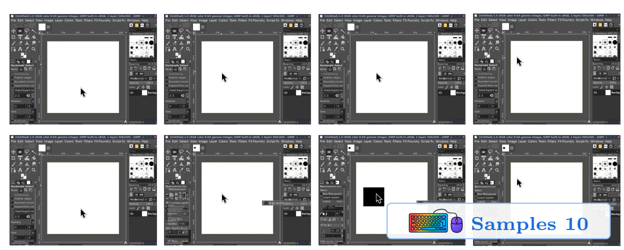


Generation
First frame and actions as input

Claude
Collect Ground Truth Video with CUA

"thinking": "Now I'll click and drag to create a selection for the square:"
"action": "left_click_drag", "x": 400, "y" : 400

Ground Truth

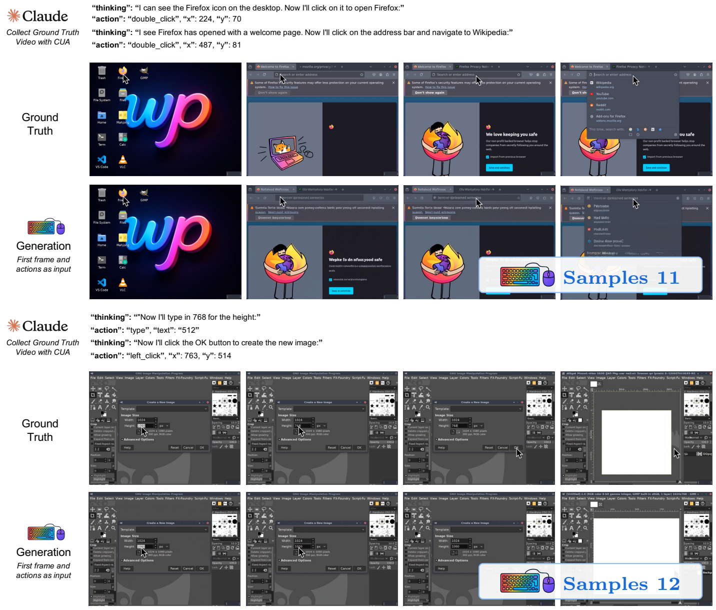


Generation
First frame and actions as input

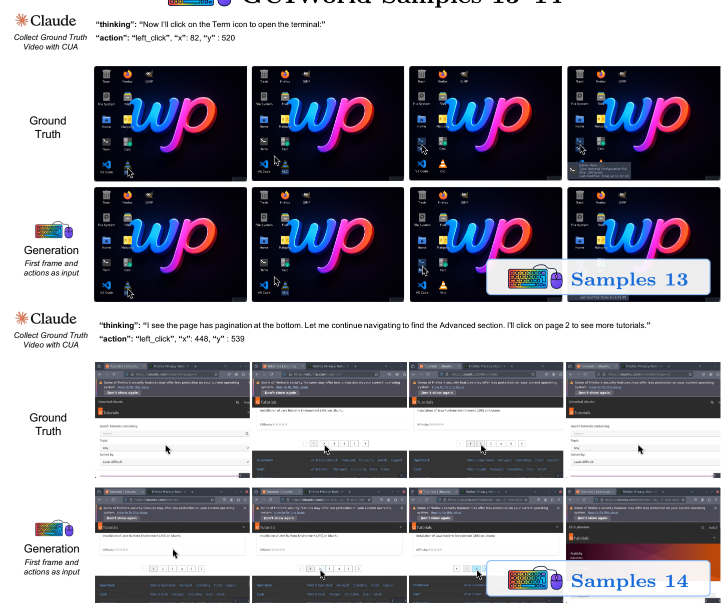
GUIWorld Visualization Thumbnails

Click any thumbnail to jump to its full-resolution page in Appendix

GUIWorld Samples 11–12



GUIWorld Samples 13–14



4 Position: Toward Completely Neural Computers

Section Overview In this section, we ask what current Neural Computer (NC) prototypes have already shown, what still prevents them from becoming usable or general-purpose runtimes, and why neither current world models (Ha and Schmidhuber, 2018; OpenAI, 2024; Polyak et al., 2024; Google DeepMind, 2025) nor AI agents (Hong et al., 2023; Anthropic; Zhuge et al., 2026a) yet amount to this emerging machine form. We then contrast NCs with conventional computers, clarifying that they are not a smarter layer on top of the existing stack, and define their mature general-purpose form, namely Completely Neural Computers (CNCs). Finally, we outline a roadmap toward CNCs, relate NCs to other system objects, and close with several remarks on NCs.

4.1 From Neural Computers to Completely Neural Computers

Current Status of NCs Our CLI and GUI-based neural computers already show that early runtime primitives can be learned with measurable interface fidelity. In terminal environments, OCR-based text fidelity is already measurable (Table 4); in GUI settings, explicit visual supervision yields strong local cursor control (Table 9); and in GUIWorld, aligned goal-directed data clearly outperforms much larger random exploration (Table 8). Taken together, these results suggest that current NCs already support early runtime primitives, especially I/O alignment and short-horizon control, while stable reuse and general-purpose execution remain out of reach. This does not mean that current prototypes are already close to CNCs; it means that the outline of a distinct machine form has begun to emerge at prototype scale.

However, the current video-based prototypes are only early NC instantiations: if NCs are to mature into general-purpose runtimes, they must go well beyond basic I/O and short-term execution. At the formal level, this ultimately requires Turing completeness (Gödel, 1931; Church, 1935; Turing et al., 1936; Siegelmann and Sontag, 1992), universal programmability (Von Neumann, 1993; Wilkes, 1981), and behavior consistency unless explicitly reprogrammed (Queloz, 2025). Before those conditions are met in full, progress is better read through practical acceptance lenses: routine reuse, execution consistency, and explicit update governance. These lenses matter because the immediate question is not whether CNCs have already been achieved, but whether NCs are beginning to behave more like usable runtimes than isolated demonstrations. For example, once an incident-response routine has been installed, the system should reuse it on later alerts rather than rediscovering the procedure from scratch each time; and if its behavior changes, that change should be attributable to an explicit update rather than ordinary execution. In practice, this reduces to three acceptance lenses: install–reuse, execution consistency, and update governance, which together offer a more useful view of current NC progress than the full CNC definition alone.

While certain sequential neural architectures are Turing complete (Siegelmann and Sontag, 1992; Pérez et al., 2021) in principle, turning a trained instance into a reliably programmable runtime remains challenging. Preliminary attempts, including Neural Virtual Machine (Katz et al., 2019) and NeuroLISP (Davis et al., 2022), have been explored. Furthermore, ensuring stable behavior over long temporal horizons remains an open problem in neural systems (Kirkpatrick et al., 2017; Calanzone et al., 2025). Section 4.2 provides a more detailed discussion of these requirements. To the best of our knowledge, existing works on world models lack an analysis of the computability class of the learned models. See Section 4.3 for more discussion between NCs and other system objects, including world models and AI agents.

Fundamental differences between NCs and conventional computers We compare NCs and conventional computers. Here, *conventional computers* denote random-access machines with instruction set architecture (Hartmanis and Simon, 1974; Anagnostopoulos et al., 1973) and layered OS/application stacks programmed via human-designed high-level languages (Backus et al., 1957, 1963). NCs differ fundamentally from conventional computers in their architectural and programming-language semantics.

At the architectural level, random-access machines instantiate local, compositional symbolic semantics (Newell and Simon, 1976), yielding exactness and interpretability, but brittleness under noise and model mismatch (Brooks, 1991). Neural computers, by contrast, realize holistic, distributed numerical semantics, trading precise local semantics for robustness and generalization (Ivakhnenko and Lapa, 1965; Ivakhnenko et al., 1967; Ivakhnenko, 1968, 1971; Hinton et al., 1986; Bishop, 2006). Empirical evidence indicates that such

Table 13 Four system objects compared at a common systems level.

System object	Organized around	Source of truth	Primary role
Conventional computer	Explicit programs	Explicit programs and explicit machine state	Reliably execute explicit programs
AI agent	Tasks	External environments, tools, and workflow state	Accomplish tasks through an existing software stack
World model	Environment dynamics	A learned model of state evolution	Predict and roll out how an environment may evolve
Neural computer	Runtime	Installed capabilities and runtime state inside the learned system	Sustain execution, accumulate capability, and govern updates within one learned machine

holistic numerical representations are particularly well suited to domains characterized by high-dimensional representations (Bengio et al., 2013), soft or statistical constraints (Smolensky, 1988), and globally coupled structures (Silver et al., 2017; Vaswani et al., 2017), including perception, natural language, planning under uncertainty, and approximate reasoning. Although conventional computers can, in principle, emulate NCs, doing so often introduces unnecessary conceptual and engineering complexity when the target tasks are already well matched to neural architectures.

At the programming-language level, NCs differ from conventional computers because their "language semantics" are the meanings of user input sequences learned from data rather than explicitly designed by humans. For example, LLMs can be viewed as programmable computers in which prompts act as programs (Reynolds and McDonell, 2021). In this case, the programming language is a natural language, which no non-neural system has historically been able to interpret robustly at scale (Jurafsky and Martin, 2026). More broadly, learned programming-language semantics are not constrained by a human-specified syntax/semantics boundary and can, therefore, encode task-relevant conventions implicitly (Wei et al., 2023).

Definition of Completely Neural Computers We use CNC to denote the mature form of an NC. Formally, a Neural Computer instance is complete if it is (1) Turing complete, (2) universally programmable, (3) behavior-consistent unless explicitly reprogrammed, and (4) realizes the architectural and programming-language advantages of NCs relative to conventional computers. The following section unpacks these conditions in operational terms.

Table 14 Operational reading of the four CNC requirements.

CNC requirement	Plain reading	What engineering evidence should look like
Turing complete	The system is not restricted to a narrow family of fixed tasks, but can in principle express general computation.	As effective memory and context grow, the same NC should remain able to carry longer and more structured procedures rather than failing by a different shortcut each time.
Universally programmable	Inputs should not only trigger one-off behavior, but install routines or internal executors that remain callable later.	Capabilities can be installed, invoked, composed, and retained across tasks rather than being relearned or outsourced each time.
Behavior-consistent	Ordinary use should not silently change the machine; behavioral change should come from explicit updates.	Same-version behavior is reproducible; execution and update traces can be inspected, replayed, and rolled back; long-horizon drift is measurable and governable.
Machine-native semantics	The system should not merely imitate conventional computers with neural components, but develop its own machine semantics and programming interfaces.	Composition, routing, continuous state, and internal executors yield usable system-level advantages; prompts, demonstrations, traces, and constraints begin to function as programming interfaces rather than mere logs.

4.2 A Roadmap Towards CNC

We frame the path toward CNCs through a set of formal requirements together with the practical challenges that must be resolved before those requirements become engineerable.

Turing completeness A Neural Computer (NC) instance (a specific architecture with fixed learned weights) defines a class of computational models in which each model corresponds to at least one memory state instance. In the formal computability discussion below, "memory state" is used in the classical state-machine sense; operationally, it corresponds to the NC runtime state introduced earlier. An NC instance is Turing complete if, for any given Turing machine, there exists an initial memory state that allows the NC to emulate that machine exactly. Notice that although Recurrent Neural Networks (RNNs), Neural Turing Machines (NTM) (Graves et al., 2014), and Differentiable Neural Computers (DNC) (Graves et al., 2016) are Turing complete in the asymptotic sense, a particular RNN, NTM, or DNC instance with finite precision cannot be Turing complete due to their fixed finite memory size. For an NC instance to achieve universality, unbounded effective memory is necessary. An NC instance has unbounded effective memory if there are infinitely many possible memory state instances. Existing works approach such unboundedness by progressively growing model parameters (Rusu et al., 2016) or context (Vaswani et al., 2017).

Universal programmability An NC is universally programmable if, for each given Turing machine, there exists an input sequence such that the NC taking this input realizes a new memory state representing the given machine. Most existing universal programmability results for neural networks are established by constructing computational primitives and proving that their composition can simulate a universal computational model (Reed and De Freitas, 2015). Likewise, we believe that universal programmability in NCs can be achieved through compositional neural programs (Pierrot et al., 2019).

Behavior consistency A CNC must preserve its function unless explicitly reprogrammed. For each memory state, there must be a non-empty set of inputs that executes the CNC without changing its pure function. Operationally, this requires a separation between run and update: ordinary inputs should execute installed capability without silently modifying it, while behavior-changing updates should occur explicitly through a programming interface. This in turn motivates training and architectural mechanisms that disentangle function use from function update, so that routines can be installed, executed, and composed without accidental functional drift. We hypothesize that gating mechanisms, such as those in LSTM (Hochreiter and Schmidhuber, 1997), are effective in achieving this conditional invariance. In practice, making this separation reliable requires clear boundaries around what state persists across tasks, what counts as an explicit update, and what execution evidence can be replayed, compared, or rolled back.

Run / update contract.

- **Run:** invoke installed capability without silently changing persistent behavior.
- **Update:** any behavior-changing modification should occur explicitly through a programming interface.
- **Required boundaries:** state (what persists), update (what counts as reprogramming), and evidence (what can be replayed, compared, or rolled back).

Architectural semantics Since NC behavior is governed by real-valued parameters, learning can produce input–output mappings that generalize across variations within the training distribution (Poggio et al., 2019). For example, after observing many instances of how the visual state of a spreadsheet interface changes when values are typed into cells, a model may learn the underlying transformation and correctly predict the screen updates for previously unseen spreadsheets that follow the same interaction rules. Such in-distribution generalization arises from the smooth function approximation properties of neural networks and their ability to interpolate across previously observed patterns. Furthermore, learning can also produce novel input–output mappings that are not explicitly represented in the training data, potentially introducing new computational primitives (Ha et al., 2016). The combination of such newly formed primitives could enable qualitatively new functions, yielding out-of-distribution functional generalization (Lake and Baroni, 2018).

Beyond emulating conventional computers, NCs can natively support functions whose semantics are ill-suited to symbolic APIs (Marcus, 2018), including probabilistic inference over high-dimensional latent states (Kingma and Welling, 2013), representation learning (Bengio et al., 2013), retrieval over dense memories (Graves et al., 2016), and end-to-end differentiable pipelines that couple perception and control (Silver et al., 2017). These functions are first-class at the architectural level and operate directly on distributed states. This enables capabilities such as learned heuristics (Silver et al., 2017), uncertainty-aware decision-making (Clements et al., 2019), and continual adaptation (Parisi et al., 2019).

Because the memory state of an NC/CNC is numerical, computer configuration and design emerge as alternatives to application-level programming: the computer itself is configured by optimizing its internal state to achieve desired computational behaviors under task-defined objectives. Depending on the differentiability of the loss, methods such as Adam (Adam et al., 2014) and natural evolution strategies(Wierstra et al., 2014) apply. In a CNC, the memory constitutes a continuous manifold, so realizing a target capability amounts to synthesizing a machine configuration (a memory state) that minimizes a user-specified loss (e.g., "minimize proof error") via direct numerical updates to the computer's state. This reframes system construction from discrete code authoring to differentiable configuration of the computer itself (Innes et al., 2019), with progress evaluated by solver convergence, stability, and reliability relative to combinatorial program search (e.g., LLM-based code generation (Hong et al., 2023)).

Programming-language semantics The learned programming-language semantics of NCs enable a shift from rigid coding to learned specifications, in which user inputs themselves function as programs (Brown et al., 2020; Wei et al., 2022). Rather than centering development on explicitly authored code, NCs expose a learned language whose syntax and semantics are acquired from data (Radford et al., 2019; Bommasani, 2021), so natural-language instructions, examples, and constraints serve as executable specifications (Ouyang et al., 2022). Consequently, brief user inputs can replace long sequences of low-level actions. Development, therefore, moves from code authoring to curating, specifying, and verifying inputs under a learned programming-language semantics, aligning system behavior with human intent via in-context specification rather than forcing users to conform to rigid, brittle interfaces (Austin et al., 2021). This does not imply that code disappears, but rather that code becomes one installation medium among several, alongside prompts, demonstrations, trajectories, and constraints.

Since NCs are programmed via users' input sequences under learned programming-language semantics, the training data for programming NCs, i.e., paired user I/O traces (Flener and Schmid, 2008), is far more abundant and continuously generated than high-quality, human-written code. Every interaction with digital systems produces structured streams of inputs, interface states, and effects that can be logged at scale (e.g., keystrokes, cursor trajectories, screen transitions), yielding orders-of-magnitude more supervision than curated program corpora (Yao et al., 2022a). These I/O traces constitute executable specifications, revealing user intentions and computer behavior (Cypher and Halbert, 1993). This enables end-to-end learning of interface conventions, control policies, and task semantics without requiring explicit program text (Yao et al., 2022b). This asymmetry in data availability favors NC training regimes that leverage ubiquitous interaction logs and, by supporting broader task coverage, reduces dependence on brittle, sparsely available code datasets.

4.3 Relations to Other System Objects

Figure 9 summarizes the systems-level shift: conventional computers are used directly, AI agents mediate existing computers, world models act as a parallel predictive layer, and NCs aim to make the learned runtime itself the machine.

The comparisons below unpack this shift relative to conventional computers, world models, and AI agents.

Conventional Computers Conventional computers remain the reference system object for reliable execution, explicit programmability, and mature governance. NCs differ not by adding a smarter application layer on top of this substrate, but by shifting computation, memory, and I/O into a learned runtime state. In this sense, NCs are best viewed as a different candidate machine form and computing substrate rather than as a direct extension of the conventional software stack. This framing does not imply that conventional computers will disappear soon, but rather that future systems may be built from a different underlying runtime substrate.

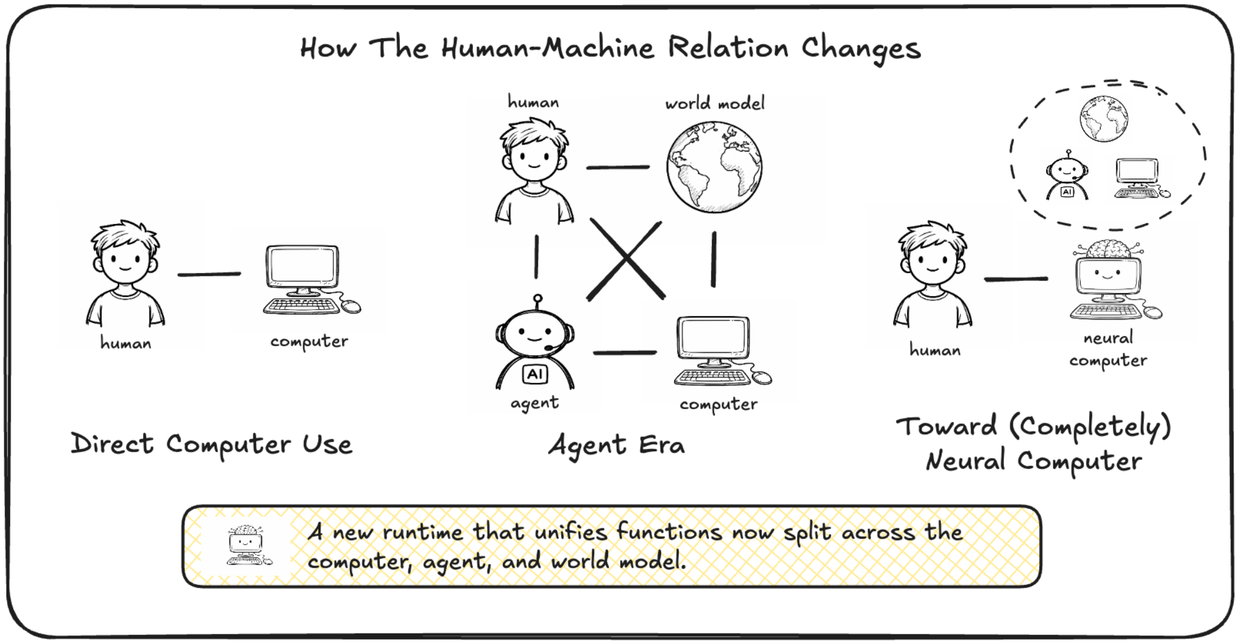


Figure 9 Changing human-machine relations across conventional computers, the agent era, and neural computers. Conventional computers are used directly; in today's agent stack, agents mediate existing computers while world models serve as a parallel predictive layer; NCs aim to unify these split functions within one learned runtime. In this sense, NCs are motivated not by replacing the current stack from the outside, but by internalizing its split functions within one learning machine. A Completely Neural Computer (CNC) is the mature, general-purpose realization of this machine form.

World Models World models learn environment dynamics by predicting action-conditioned transitions (Ha and Schmidhuber, 2018). Such target environments range from the most ambitious, where all sensory inputs form the real world (Schmidhuber, 1990), to much narrower scopes, such as a few control parameters of a robot arm (Feng et al., 2023). They provide one technical perspective on current NC prototypes, since interactive computers are an important class of action-conditioned environments, but they do not by themselves define the NC abstraction. Many current approaches to modeling computational environments, such as the physical world, also rely heavily on computer-generated data (Richter et al., 2016; Tobin et al., 2017; Dosovitskiy et al., 2017; Garcia et al., 2023), potentially leading to models that share characteristics with neural computers.

AI Agents Another important comparison point is AI agents built on top of modern AI models and external software substrates, including computer-use agents, coding and multi-agent systems (OpenAI, 2023; Anthropic; Hong et al., 2023; Zhuge et al., 2024a; Sager et al., 2025), and recursive self-improvement loops (Zhuge et al., 2026b). These systems place a learned agent between the user and an external execution substrate, whether that substrate is a GUI, a codebase, or a broader software toolchain. This provides strong leverage from existing computers and software stacks, but it also preserves a strict separation between the learned model and the runtime that actually stores executable state, applies updates, and enforces system contracts. Computer-use agents operate through low-bandwidth I/O; coding agents typically emit symbolic artifacts that must be executed elsewhere; and RSI-style loops improve the agent by iterating over external tools, prompts, or code rather than by turning the runtime itself into the computer. Such systems also increasingly rely on automated evaluators, including agent-as-a-judge schemes, to rank outputs, validate task completion, and close iterative improvement loops (Zhuge et al., 2024b). We hypothesize that a sufficiently capable NC can internalize many of these agentic functions within one persistent neural runtime.

4.4 Additional Thoughts

The remarks in this section are intended as hypotheses and design directions motivated by the present results, rather than as empirical conclusions established by the current prototypes.

ONE ONE (Schmidhuber, 2018) proposed a single neural substrate that incrementally absorbs and reuses diverse learned skills. While ONE was not instantiated as a computer-like runtime with explicit I/O, programmability, and update governance, a mature CNC can be viewed as a plausible systems-level realization of this idea. In this sense, many specialized world-model-like components may ultimately appear not as separate external systems, but as installable capabilities within one persistent neural runtime.

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 state 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 5). 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 linear/quadratic Transformers (Schmidhuber, 1992; 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. Developing discrete operations, compositional structures, and verifiable computation that are harmonious in neural systems may play an essential role in designing such systems. This approach follows more closely the construction of conventional computers from well-defined computational primitives and stands in contrast to relying on emergent reasoning in generic video generation models.

Neural networks generation via NC interaction Neural network generation can be viewed as a form of programming, i.e., the synthesis of a neural architecture and its corresponding weights. Because NCs' architectural semantics are already neural and numerical, neural components are first-class, and generation directly manipulates the memory state rather than translating it into symbolic code. Moreover, NCs can be programmed through I/O interaction: sequences of inputs, observations, and outcomes act as executable specifications that shape the internal state and routines of the system (Cypher and Halbert, 1993; Myers, 2002). This suggests a path in which users generate and refine neural modules within NCs through interactive traces, treating interaction logs as programs that configure and compose neural computation.

Unified hardware requirements and data representation In NCs, tensors and tensor-to-tensor transformations act as primary computational primitives, replacing the heterogeneous mix of data structures and subsystem-specific abstractions common in conventional computers. Traditional systems span many distinct domains—scalars, pointers, linked structures, files, sockets, and processes—each with its own memory layout, invariants, APIs, and failure modes, coordinated by operating systems through largely disjoint subsystems (virtual memory, filesystems, networking, scheduling, and drivers) (Tanenbaum and Austin, 2013; Silberschatz et al., 2019). Although this heterogeneity supports broad generality, it also fragments optimization and tooling because compilers, profilers, and debuggers must reason across incompatible abstractions (Gregg, 2014). By contrast, a tensor-uniform pipeline concentrates representation and execution into a compact set of composable primitives, such as linear algebra and elementwise operations, allowing tooling to target a shared intermediate representation (Paszke et al., 2019). As a result, optimizations such as operator fusion, memory planning, and computational-graph rewriting can be applied system-wide (Vasilache et al., 2018); profiling can focus on throughput and memory bandwidth; and accelerators such as GPUs can be targeted through common tensor runtimes (Sze et al., 2017). This shared numerical representation also naturally supports multimodal computation: vision (pixel tensors), language (sequence embeddings), audio (waveforms or spectrograms), control (state-action tensors), and planning (latent trajectory tensors) all reside in one representational space and can be jointly reasoned over and optimized in a single graph (Ramachandram and Taylor, 2017), without repeated type bridging or subsystem translation—steps that are substantially harder in traditional heterogeneous stacks.

5 Conclusion

Neural computers point toward a machine form in which a single latent runtime state acts as the computer itself, driving pixels, text, and actions while subsuming what operating systems and interfaces handle today. In this paper, the main result is that NCs have begun to exhibit early runtime primitives—most notably I/O alignment and short-horizon control—while stable reuse, symbolic reliability, and runtime governance remain unresolved. Our CNC capability map remains useful as a longer-horizon view, spanning efficiency, computation & reasoning, memory & storage, I/O & control, tool bridges, condition-driven generalization, programmability, and artifact generation. The map is staged and dependency-informed, but the more immediate gap is still the gap from prototype behavior to usable runtime behavior. Progress toward CNCs will therefore depend not only on stronger models, but also on whether reuse, consistency, and governance become sustained and testable. If these gaps continue to close, neural computers will look less like isolated demonstrations and more like a plausible candidate machine form for next-generation computers.

Acknowledgements

The authors sincerely thank Yasheng Sun for his early input of the GUI data collection. The authors thank Deyao Zhu and Firas Laakom for their feedback on the manuscript. Mingchen Zhuge, Haozhe Liu, Shuming Liu, Wenyi Wang, Wenxuan Zhang, Junjie Fei, and Jürgen Schmidhuber were supported by funding from the King Abdullah University of Science and Technology (KAUST) Center of Excellence for Generative AI (award number 5940) and the SDAIA-KAUST Center of Excellence in Data Science and Artificial Intelligence.

References

Kingma DP Ba J Adam et al. A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 1412(6), 2014.

Paul Constantine Anagnostopoulos, MJ Michel, Gary H Sockut, George M Stabler, and Andries van Dam. Computer architecture and instruction set design. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 519–527, 1973.

Anthropic. Computer use tool — platform.claude.com. https://platform.claude.com/docs/en/agents-and-tools/tool-use/computer-use-tool. [Accessed 02-02-2026].

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

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

John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, Harold Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

Christopher M Bishop. *Pattern Recognition and Machine Learning*, volume 4 of *Information science and statistics*. Springer, 2006. ISBN 9780387310732. doi: 10.1117/1.2819119. http://www.library.wisc.edu/selectedtocs/bg0137.pdf.

Rishi Bommasani. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1):139–159, 1991. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(91)90053-M. https://www.sciencedirect.com/science/article/pii/000437029190053M.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

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.

A. Church. An unsolvable problem of elementary number theory. *Bulletin of the American Mathematical Society*, 41: 332–333, 1935.

William R Clements, Bastien Van Delft, Benoît-Marie Robaglia, Reda Bahi Slaoui, and Sébastien Toth. Estimating risk and uncertainty in deep reinforcement learning. *arXiv preprint arXiv:1905.09638*, 2019.

Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

Gregory P Davis, Garrett E Katz, Rodolphe J Gentili, and James A Reggia. Neurolisp: High-level symbolic programming with attractor neural networks. *Neural Networks*, 146:200–219, 2022.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

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.

Yunhai Feng, Nicklas Hansen, Ziyan Xiong, Chandramouli Rajagopalan, and Xiaolong Wang. Finetuning offline world models in the real world. *arXiv preprint arXiv:2310.16029*, 2023.

Pierre Flener and Ute Schmid. An introduction to inductive programming. *Artificial Intelligence Review*, 29(1):45–62, 2008.

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.

Ricardo Garcia, Robin Strudel, Shizhe Chen, Etienne Arlaud, Ivan Laptev, and Cordelia Schmid. Robust visual sim-to-real transfer for robotic manipulation. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 992–999. ieee, 2023.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

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

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.

Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

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

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

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*, pages 2555–2565. PMLR, 2019b.

Juris Hartmanis and Janos Simon. On the power of multiplication in random access machines. In *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, pages 13–23. IEEE, 1974.

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.

G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. *Distributed representations*, page 77–109. MIT Press, Cambridge, MA, USA, 1986. ISBN 026268053X.

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.

Alekseĭ Grigorevich Ivakhnenko and Valentin Grigorévich Lapa. *Cybernetic Predicting Devices*. CCM Information Corporation, 1965.

Aleksey Grigorievitch Ivakhnenko. The group method of data handling – a rival of the method of stochastic approximation. *Soviet Automatic Control*, 13(3):43–55, 1968.

Aleksey Grigorievitch Ivakhnenko. Polynomial theory of complex systems. *IEEE Transactions on Systems, Man and Cybernetics*, (4):364–378, 1971.

Aleksey Grigorievitch Ivakhnenko, Valentin Grigorievitch Lapa, and Robert N McDonough. *Cybernetics and forecasting techniques*. American Elsevier, NY, 1967.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, with Language Models*. 3rd edition, 2026. https://web.stanford.edu/~jurafsky/slp3/. Online manuscript released January 6, 2026.

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.

Garrett E Katz, Gregory P Davis, Rodolphe J Gentili, and James A Reggia. A programmable neural virtual machine based on a fast store-erase learning rule. *Neural Networks*, 119:10–30, 2019.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

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.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.

Gary Marcus. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018.

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

Brad A Myers. Demonstrational interfaces: A step beyond direct manipulation. *Computer*, 25(8):61–73, 2002.

Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Commun. ACM*, 19 (3):113–126, March 1976. ISSN 0001-0782. doi: 10.1145/360018.360022. https://doi.org/10.1145/360018.360022.

OpenAI. Computer-Using Agent — openai.com. https://openai.com/index/computer-using-agent/, 2023. [Accessed 07-02-2026].

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.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

Thomas Pierrot, Guillaume Ligner, Scott E Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. *Advances in Neural Information Processing Systems*, 32, 2019.

Tomaso Poggio, Andrzej Banburski, and Qianli Liao. Theoretical issues in deep networks: Approximation. *Optimization and Generalization*, 2019.

Adam Polyak, Amit Zohar, Andrew Brown, Andros Tjandra, Animesh Sinha, Ann Lee, Apoorv Vyas, Bowen Shi, Chih-Yao Ma, Ching-Yao Chuang, et al. Movie gen: A cast of media foundation models. *arXiv preprint arXiv:2410.13720*, 2024.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

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*, pages 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.

Dhanesh Ramachandram and Graham W Taylor. Deep multimodal learning: A survey on recent advances and trends. *IEEE signal processing magazine*, 34(6):96–108, 2017.

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.

Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI EA '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380959. doi: 10.1145/3411763.3451760. https://doi.org/10.1145/3411763.3451760.

Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

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.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Pascal J. Sager, Benjamin Meyer, Peng Yan, Rebekka von Wartburg-Kottler, Layan Etaiwi, Aref Enayati, Gabriel Nobel, Ahmed Abdulkadir, Benjamin F. Grewe, and Thilo Stadelmann. A comprehensive survey of agents for computer use: Foundations, challenges, and future directions, 2025. https://arxiv.org/abs/2501.16150.

J. Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4 (1):131–139, 1992.

J. Schmidhuber. On decreasing the ratio between learning complexity and number of time-varying variables in fully recurrent nets. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 460–463. Springer, 1993a.

J. Schmidhuber. A self-referential weight matrix. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 446–451. Springer, 1993b.

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.

Jürgen Schmidhuber. One big net for everything. *arXiv preprint arXiv:1802.08864*, 2018.

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*, pages 440–449, 1992.

Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 2019.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550 (7676):354–359, 2017.

Paul Smolensky. On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 11(1):1–23, 1988. doi: 10.1017/s0140525x00052432.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

A.S. Tanenbaum and T. Austin. *Structured Computer Organization*. Pearson, 2013. ISBN 9780132916523. https://books.google.com.sa/books?id=m0HHygAACAAJ.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

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

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

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.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. https://arxiv.org/abs/2201.11903.

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.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022b.

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*, pages 3836–3847, 2023.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024a.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024b.

Mingchen Zhuge, Ailing Zeng, Deyao Zhu, Sherry Yang, Vikas Chandra, and Jürgen Schmidhuber. Ai with recursive self-improvement. In *ICLR 2026 Workshop Proposals*, 2026a.

Mingchen Zhuge, Ailing Zeng, Deyao Zhu, Sherry Yang, Vikas Chandra, and Jürgen Schmidhuber. Ai with recursive self-improvement. In *ICLR 2026 Workshop Proposals*, 2026b.

Appendix

A Explorations: Alternative Data Sources and Online Interaction

Beyond the data collection pipelines used in the main text and Appendix B, we explored alternative data sources for neural-computer prototyping. These routes were not incorporated into the final pipeline, but the trials yielded useful insights and suggest directions for future work as tooling matures and data-collection infrastructure scales.

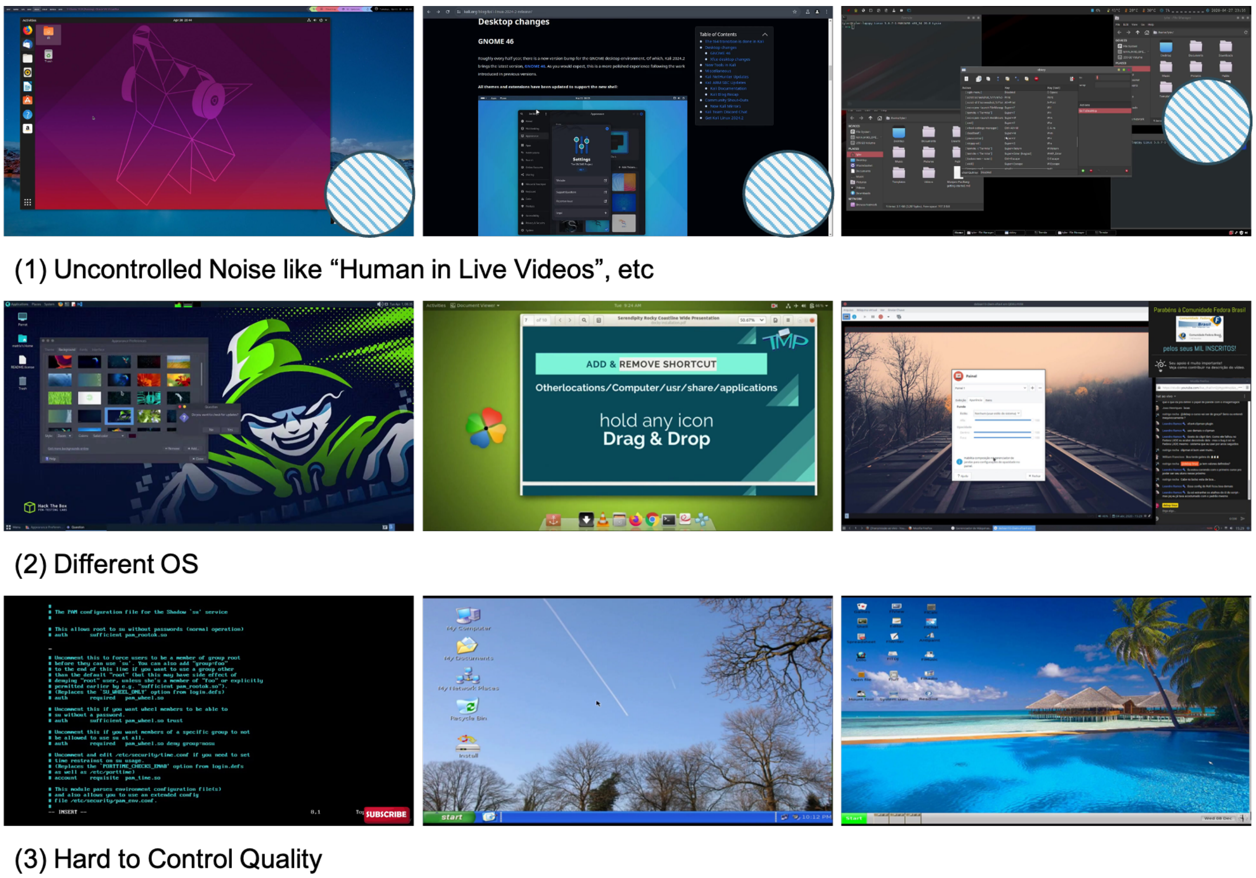


Figure 10 Common issues in web-scale screen video crawling. Web videos mix uncontrolled content, heterogeneous OS/UI configurations, and inconsistent capture quality. This makes terminal localization, OCR, and time alignment unreliable without heavy filtering and sanitization.

A.1 Web video extraction

We initially tested crawling computer-use videos from the web. We used OCR and layout detectors to locate terminal regions, estimate text content and timestamps, and extract related clips. We did not adopt this route for two main reasons:

- **Data governance constraints:** privacy and copyright risks are difficult to manage at web scale.
- **Data quality burden:** heavy filtering and sanitization are required before the data can support reliable OCR and temporal alignment.

Screen recordings can contain personal identifiers (usernames, emails, file paths, chat content) and may come with licensing constraints that are difficult to verify at scale. Cleaning the resulting data is substantially more complex than it appears (Figure 10). 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.

Despite these challenges, web videos remain a potential long-term scaling axis for interface experience. 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, rights-cleared interface trajectories. Future efforts that invest in rights-respecting acquisition and stronger automated filtering could unlock web-scale data as a complementary scaling axis.

A.2 Online environment interaction

We prototyped an agentic interaction pipeline that separates a control plane from an execution environment plane (Figure 11). In the environment plane, a sandboxed container runs a live shell together with LLM agents (planner/controller) and a recorder exposed via a narrow port-based interface. The agents issue commands and control actions. The recorder captures synchronized terminal renders, structured terminal state when available (e.g., buffer/text), and action traces. Structured terminal state is logged for diagnostics/alignment and is not fed to video models as privileged state input. Trajectories are streamed to the control plane for storage and video-model updates.

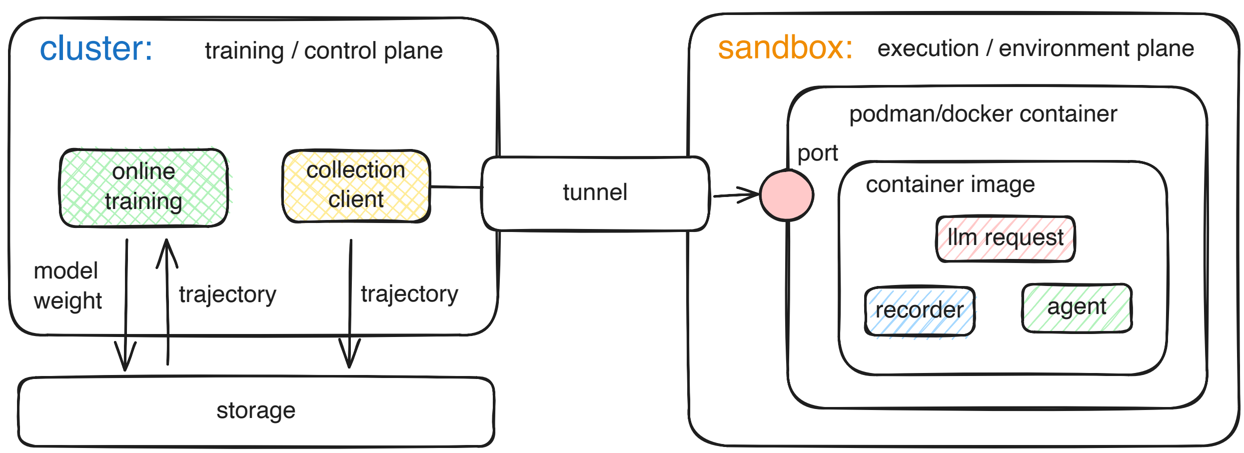


Figure 11 Agentic online interaction pipeline (early exploration). A control plane ingests trajectories for storage and video-model updates (online where feasible, with offline replay support). A sandboxed environment plane executes an agent in an isolated container and records synchronized state/action 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 rollout collection and model updates with decoupled throughput. From the perspective of Toward CNC, this design also makes the evidence boundary more explicit by preserving replayable execution traces with clear provenance across collection and training.

We explored this setup because closed-loop interaction can induce a natural curriculum by continually sampling the boundary of current video-model behavior. 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 dominant bottlenecks in the end-to-end system were systems-level: crosscluster communication latency between rollout workers and training nodes, and high debugging complexity in asynchronous distributed execution. Safety controls (e.g., isolation of untrusted code execution, monitoring and abuse prevention, and deterministic resets and environment control) remained necessary constraints, but they were not the primary bottleneck. The setup also requires robust recording and serialization across heterogeneous environments. Under time and cost constraints, we therefore prioritized controlled video data for the main experiments. Despite not being used in the final pipeline, we consider this design a useful systems template for future work. It supports scalable multi-environment rollout collection and consistent provenance and storage of trajectories. It can support multiple downstream learning algorithms (e.g., behavior cloning or preference-based learning). Execution remains sandboxed and auditable.

B 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 C. Data collection uses a three-stage pipeline to maintain synchronized timing, privacy controls, and consistent, well-documented artifacts across CLIGen and GUIWorld.

Sourcing. We construct our datasets from three complementary sources—public terminal recordings, scripted terminal replays, and a controlled desktop-capture rig. Across all sources, we emphasize rights-respecting acquisition, privacy filtering, and temporally aligned interface signals (frames, actions, and text when available):

- **CLIGen (General)**: public `asciinema .cast` archives; traces are replayed with official tools to preserve recorded terminal appearance (color schemes, cursor visibility, window geometry).
- **CLIGen (Clean)**: deterministic `vhs` scripts (e.g., package installs, REPLs, log filters) executed in isolated environments.
- **GUIWorld**: footage captured with 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. This packaging turns each episode into a replayable artifact with provenance, which is useful not only for evaluation reproducibility but also for the broader evidence and governance requirements discussed in Toward CNC.

B.1 Caption fields and metadata (CLIGen General)

For CLIGen (General), each replayed `.cast` fragment (example shown in Section C.1) is paired with three aligned descriptions and a compact metadata record. Table 15 summarizes the fields for clip `7****_0001`.

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`.

B.2 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 $T = \min(T_{\text{gen}}, T_{\text{gt}})$. If $T \leq K$, we use all indices in $[0, T{-}1]$; otherwise, we select K evenly spaced indices by deterministic rounding, then deduplicate and sort them so evaluation frames are spread across the trajectory. For every selected 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 $(g_1, \dots, g_{N_g})$ and the generated frame $(p_1, \dots, p_{N_p})$.

Table 15 Caption tiers and metadata for CLIGen (General) clip `7****_0001`.

Field	Content (abridged)
caption	The user (username field shown verbatim as annonomous [sic]) 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 0.13s, variance 0.34), and 38 visual-change events.
metadata	Source recording on Asciinema: id 7****, title "aneo.nvim demo", author annonomous [sic], creation time 2025-05-13T23:37:53Z, URLs for the page and raw .cast.

We summarize the OCR text-space metrics used per sampled frame as follows:

Character accuracy. This metric 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 $\text{char\_acc} = 1$; if only s is empty we set $\text{char\_acc} = 0$. Otherwise,

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

Extra or missing characters are normalized by the reference length through the denominator $\max(|s|,1)$. Frame-level scores are averaged over the selected frame pairs (up to $K{=}5$) to yield a per-video character accuracy, and group-level scores report the mean over videos.

Exact-line accuracy. This metric 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 $\text{exact\_line\_acc} = 1$; if the ground-truth list is empty but the generated list is not, we set $\text{exact\_line\_acc} = 0$. 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.

Together, these two metrics stress both fine-grained text fidelity and line-ordered terminal state reconstruction.

B.3 Evaluation metrics and protocol (GUIWorld)

We report both global video metrics and action-driven metrics that focus on post-interaction frames. We compute these metrics using our GUIWorld evaluation suite. We summarize the GUIWorld protocol at a glance as follows:

- **Global metrics** (FVD_{all}, SSIM_{all}, $\text{LPIPS}_{\text{all}}$): computed over paired generated/ground-truth videos after standardized decoding, subsampling, and resizing.
- **Action-driven metrics** (SSIM_{+15}, LPIPS_{+15}, FVD_{+15}): computed on post-action windows to measure interface fidelity after interaction events.

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), map each timestamp τ to a frame index $f = \text{round}(\tau \cdot \text{fps})$, and clamp to the valid frame range. We skip the action frame itself (`action_start_offset`=1) and evaluate the next k=15 frames after each action. Concretely, for each action frame index f, we form the post-action set $\{f + 1, \ldots, f + k\}$ and clip it to valid frame indices. We then take the deduplicated union over actions in the clip and keep frame indices in chronological order. Clips with zero logged actions, or with empty valid post-action windows after clipping, are excluded from +15 metrics. For SSIM_{+15} /LPIPS_{+15}, we compute per-clip means over selected frame pairs and then average across clips. For action-driven FVD_{+15}, we build an *after-action clip* per video by concatenating the same selected post-action frames, then uniformly subsample/pad to 16 frames and compute the Fréchet distance in the same `r3d18` feature space.

C CLIGen: CLI Trajectory Formats

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

C.1 Asciinema (.cast) example

The header line stores the recording config (version, terminal size, timestamp, env). In this excerpt, output rows follow `[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"]
```


C.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 prompt.
# 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
```

D GUIWorld: Action Representation, Temporal Alignment, and Conditioning

This appendix provides additional technical details on GUIWorld action representation, temporal alignment, and conditioning used in Section 3.2. Additional visualization pages are collected in Appendix E; evaluation metrics and protocols are summarized in Appendix B.3.

D.1 Action schema

$\mathrm{NC}_{\mathrm{GUIWorld}}$ 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. In all reported v2 experiments, we use S=2 action slots per frame; empty slots are padded with type 0. Each action has a type (e.g., `mouse click` or `keyboard type`) plus parameters. Table 16 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 16 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`)

D.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 16) 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, summarized below:

- `external`: fuse actions at the VAE input.
- `contextual`: mix actions and frames as tokens in one sequence.
- `internal`: inject actions inside transformer blocks.
- `residual`: add lightweight action deltas to hidden states.

Injection-mode definitions (schematic) Below we give compact schematic definitions for the three formula-based modes (`external`, `residual`, `internal`); `contextual` is specified by the structured attention mask in Figure 12. **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

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

D.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 approximately have $F \approx (T-1)c + 1$ under uniform sampling. Exact indexing follows the dataloader timestamp mapping and boundary handling. 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 lag-aware local attention mask. Appendix Figure 12 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 recent actions in a lag-bounded recent-action range. In implementation, this window is $j \in [\max(0, i-\ell), \min(i, A-1)]$, where ℓ is the action lag and A is the action-token length. This way, frame conditioning stays focused on recent operations and excludes future actions. In the lower-left block ($A2V$), an action A_i can attend to frames V_t that occur after it has had time to take effect ($t \ge i + \ell$, with boundary clipping), but not to earlier frames. This path is representation-only and does not expose future frame information to frame prediction. The lower-right block ($A2A$) is strict diagonal: each action token attends to itself.

In practice (w, ℓ) act as fixed hyperparameters that trade off temporal coverage and cost. Together, these choices implement a structured lag-aware temporal prior: actions do not explain past frames, and each frame conditions on 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, in our experiments, with the visual backbone fixed, control fidelity improved more from conditioning design than from encoder choice. 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.

D.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 recover a small, high-frequency visual target, 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. 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). Each coordinate is then mapped 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 the trajectory tensor `mouse_trajectories` used across rendering and action encoding.

D.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 image-to-video (I2V) model. For each clip we form reference images $\text{ref\_img}_{0:T-1}$ and masks $\text{ref\_mask}_{0:T-1}$:

- At $t{=}0$, ref_img_0 is the full desktop frame and ref_mask_0 is all ones, anchoring the static layout and theme;
- For $t{>}0$, ref_img_t is the cursor foreground f_t and 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 enforces masked, reference-consistent reconstruction inside the cursor region while relying on learned dynamics 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 ref_mask_t. Table 9 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

Table 17 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.

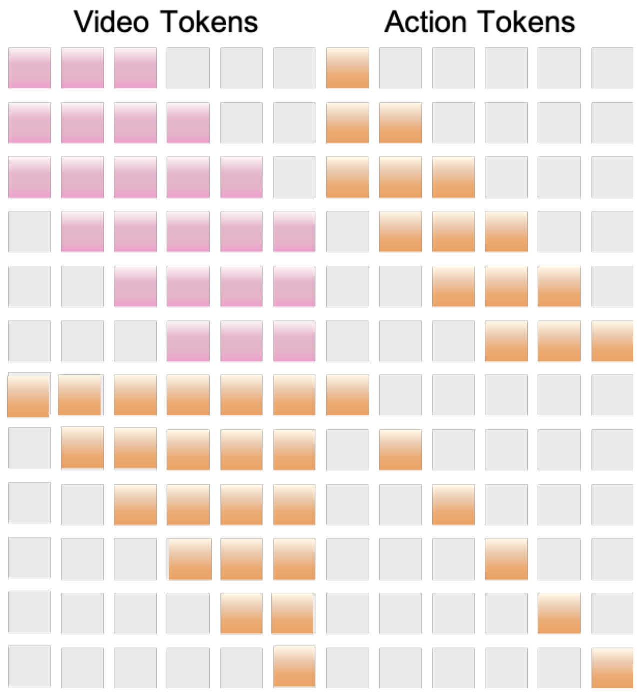


Figure 12 Contextual attention mask for GUIWorld. Video tokens (pink) and action tokens (orange) share one sequence. The mask restricts attention to a short temporal window so each frame attends only to nearby frames and temporally aligned actions.

$A_t \in \mathbb{R}^{d_a}$ 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. In implementation, matching is lag-aware: frame t is aligned with action/mouse features at $t - \ell$, where ℓ is the configured `action_lag`. Similarities are scaled by a temperature τ. It pushes matched pairs 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 $t + \ell$. 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 tight temporal 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 17 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.

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 10. Full metric definitions and the evaluation protocol are provided in Appendix B.3.

E Additional Visualizations

E.1 CLIGen Visualizations

This subsection consolidates *all* CLIGen visualization pages referenced in the paper. The main text keeps section-local thumbnail panels at the end of each visualization subsection, while full-size pages are collected here.

(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, classic shell prompts and system outputs, long-running jobs with rapidly scrolling and color-coded package logs, full-screen TUIs, and progress dashboards.

(2) CLIGen (Clean) REPL visualizations. In contrast to open-world traces, CLIGen (Clean) REPL samples are scripted and temporally well-paced (Figures 16–19; additional format examples are in Appendix C). Each sample includes an explicit action trace (e.g., `Sleep`, `Type`, `Enter`, arrow keys, `Hide`) alongside rendered terminal frames, making action-to-pixel causality easy to inspect.

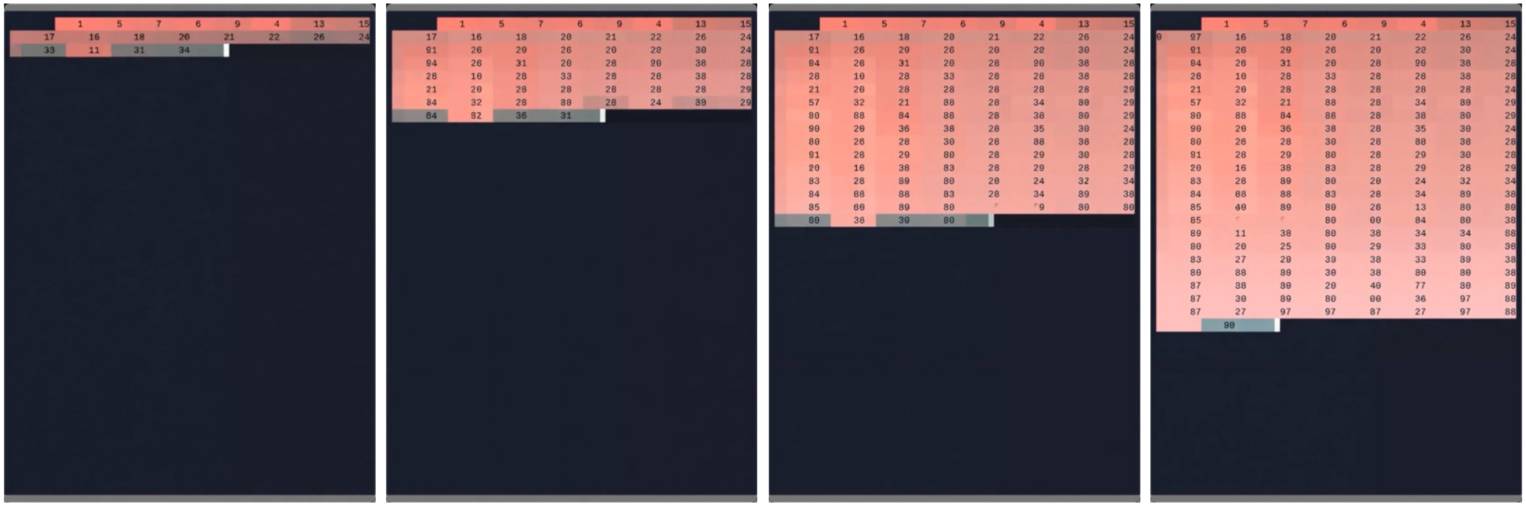
(3) CLIGen (Clean) math visualizations. Figures 20–22 compare model rollouts on CLIGen (Clean) math REPL prompts. Figures 23–25 show reprompting cases and highlight why these probes should separate native computation from answer-conditioned rendering.

E.2 GUIWorld Visualizations

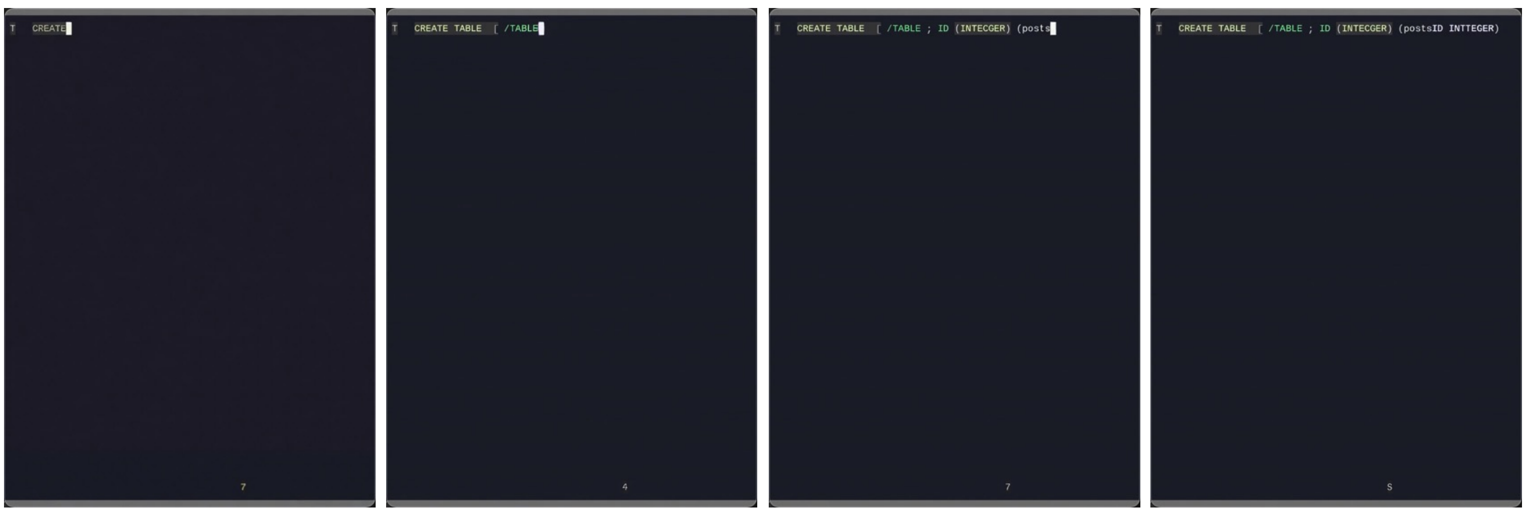
This subsection consolidates *all* GUIWorld rollout visualizations (Figures 26–39). CUA-based pages overlay the CUA trace (natural-language rationale when available, e.g., in a `thinking` field, plus structured *action* fields such as `left_click`, `double_click`, `left_click_drag`, and `type`). It contrasts the *Ground Truth* trajectory (top) with a *Generation* conditioned on the first frame and the action sequence (bottom), making state drift easy to spot.

Figures 31–33 emphasize compounding low-level deviations; Figures 34–36 focus on numeric/UI fidelity and interaction semantics; and Figures 37–39 provide additional stress cases where correctness hinges on precise field edits and page state.

Back to Main Thumbnails



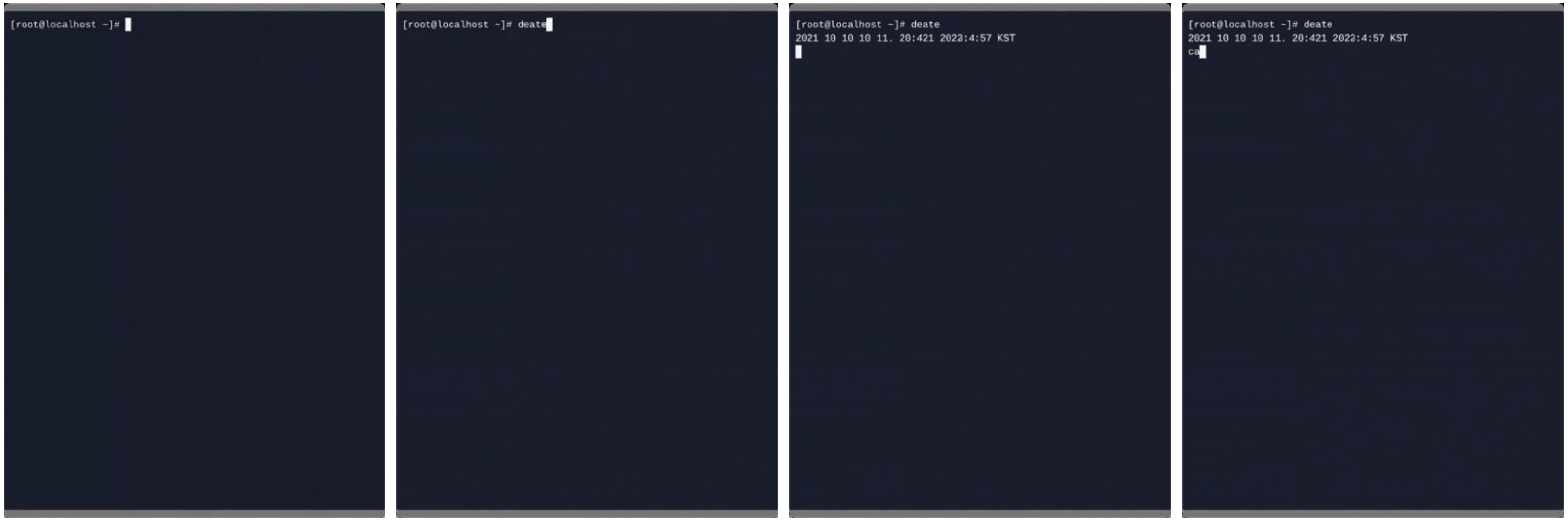
The terminal displays a series of ANSI escape code formatted texts with changing background and foreground colors, executing commands like `\\u001b[48;2;255;128;128;38;2;0;0;0m` which set the background to a shade of pink and text to black, and printing numbered lists with colors. The output includes 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.



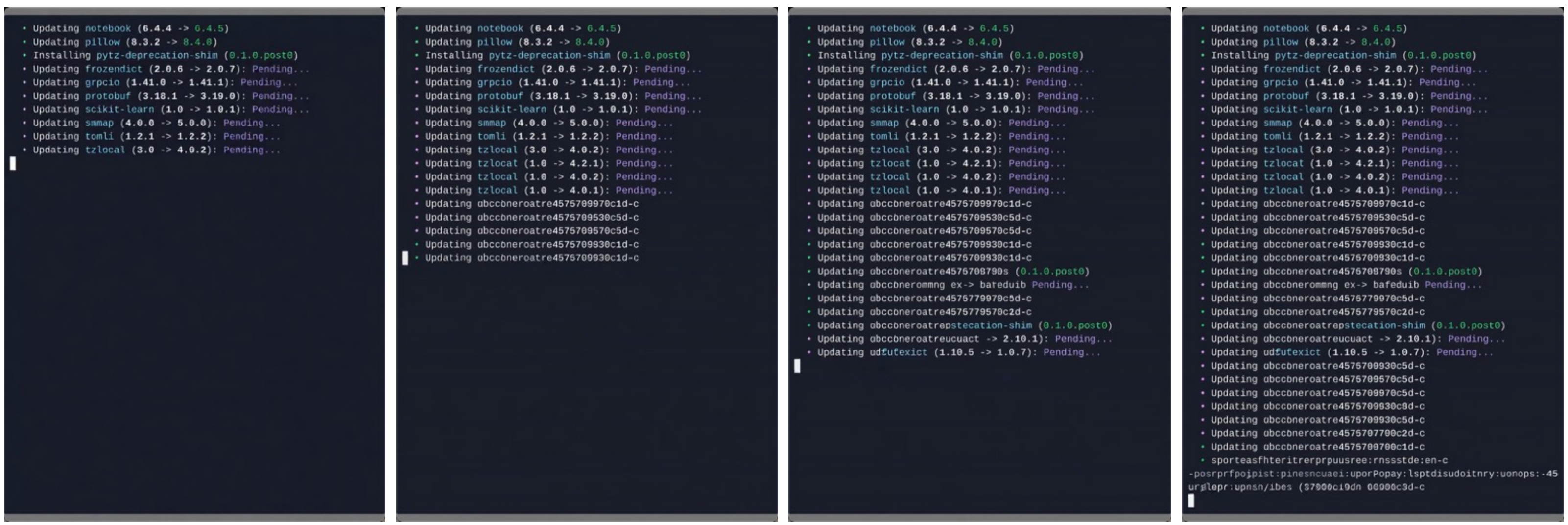
The user types the command `CREATE TABLE posts (ID INTEGER)`, with the terminal displaying the command in a dark background with colored syntax highlighting, including green and yellow text, and the cursor moving character-by-character as the user types, with some corrections and backspacing along the 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 13 CLIGen (General) visualization samples (A).

Back to Main Thumbnails



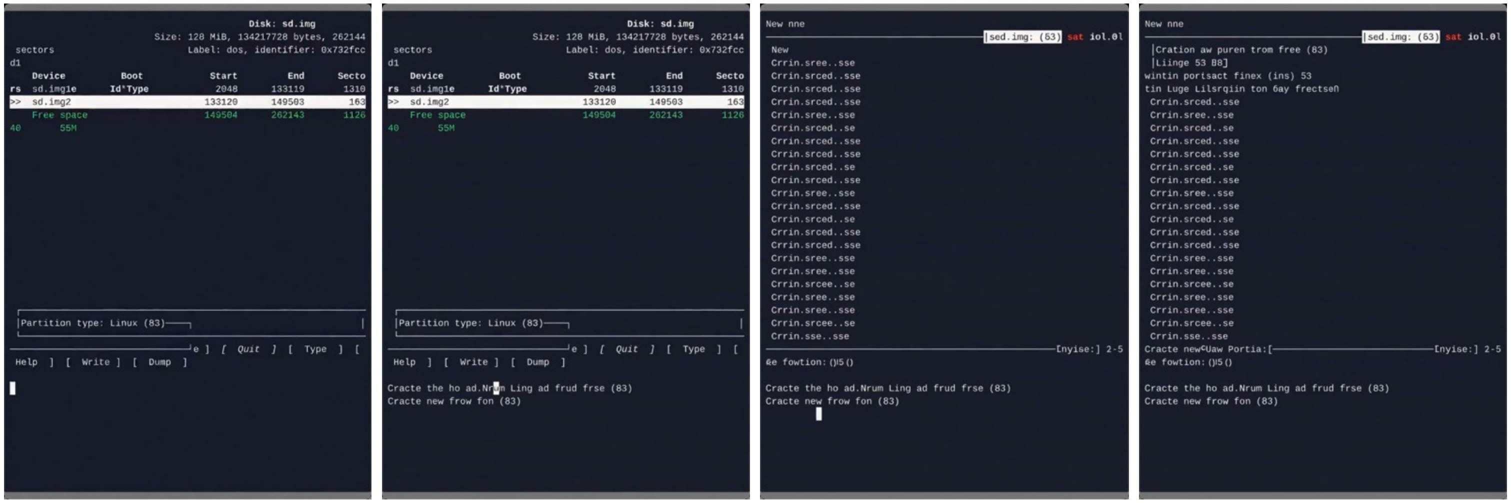
At the `root@localhost:~#` prompt, the user types the `date` command, which displays the current date and time in a plain text format as \"2021. 10. 11. 22:47:43 KST\", then begins typing the `cat` command.



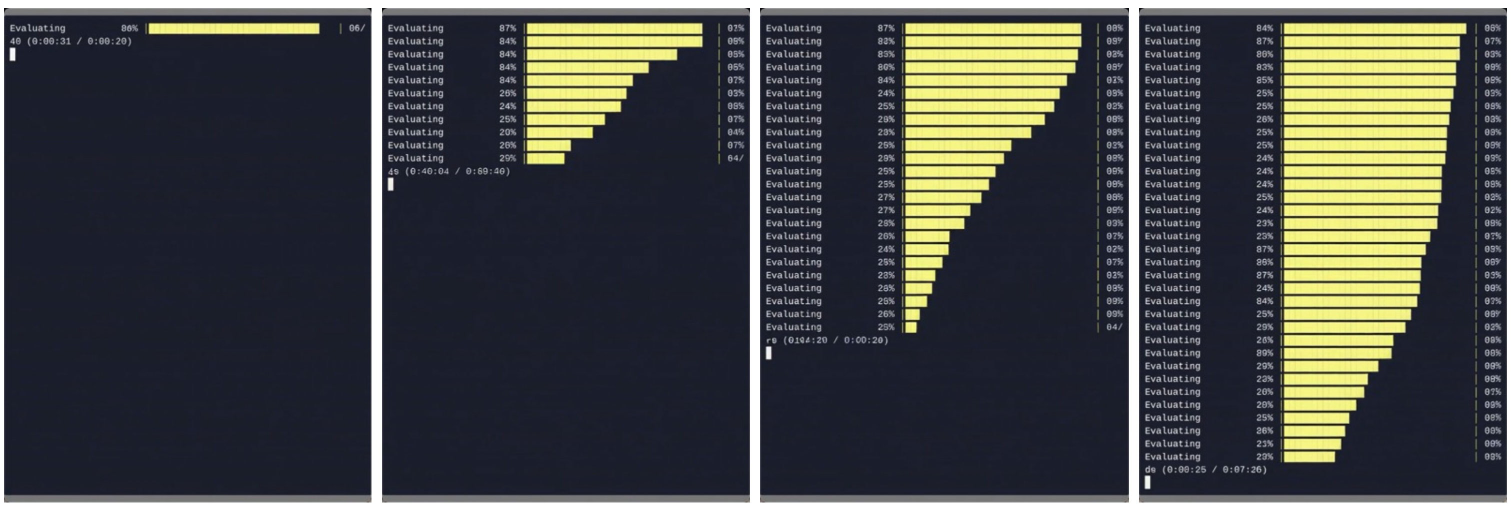
The terminal displaying progress bars, package names like `pillow`, `notebook`, and `tzlocal`, and version changes in green and red text. The output shows downloading and installing statuses, including percentages, for packages like `smmap`, `tomli`, and `protobuf`, with the terminal scrolling through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

Figure 14 CLIGen (General) visualization samples (B).

Back to Main Thumbnails



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 15 CLIGen (General) visualization samples (C).

Back to Main Thumbnails

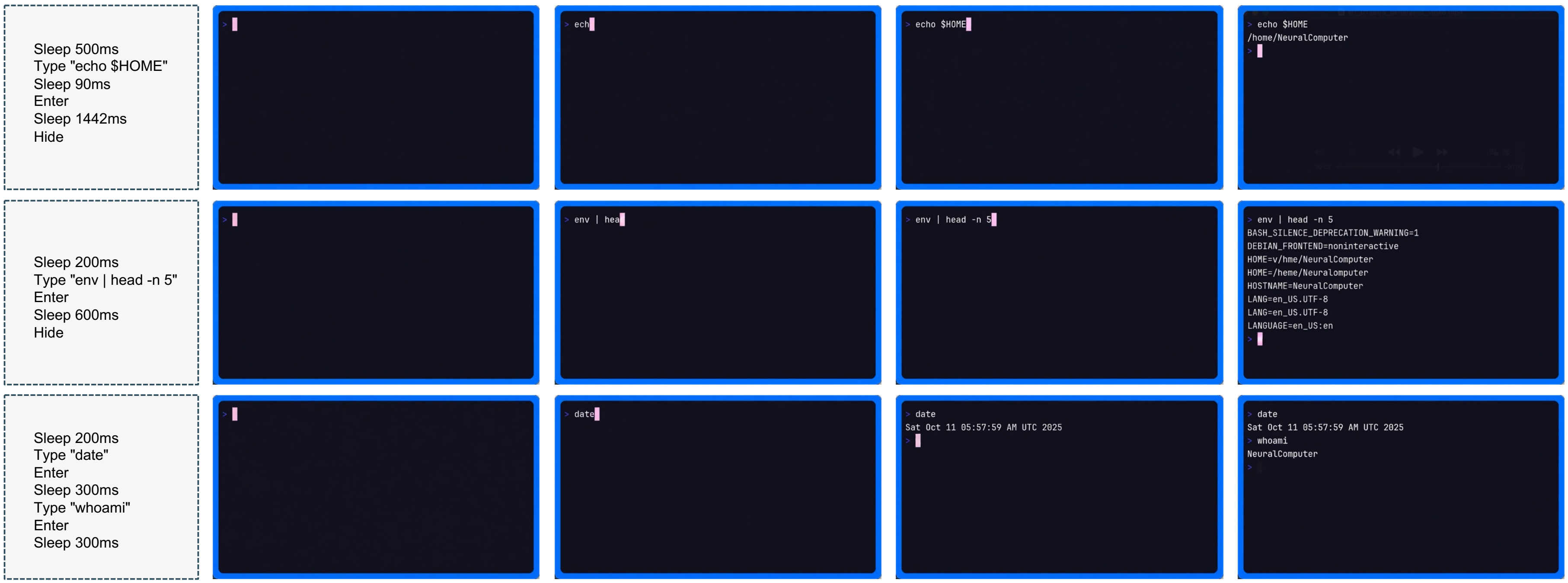


Figure 16 CLIGen (Clean) REPL visualization samples (A).

Back to Main Thumbnails

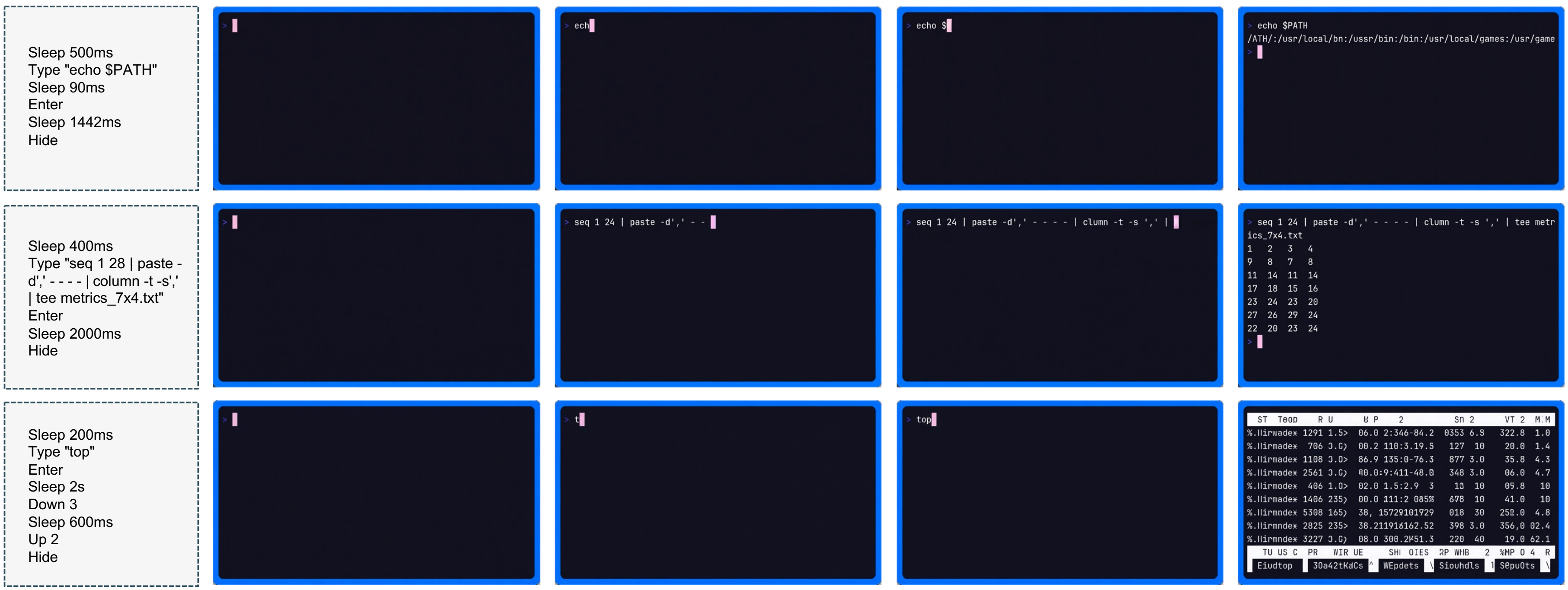


Figure 17 CLIGen (Clean) REPL visualization samples (B).

Back to Main Thumbnails

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

Back to Main Thumbnails

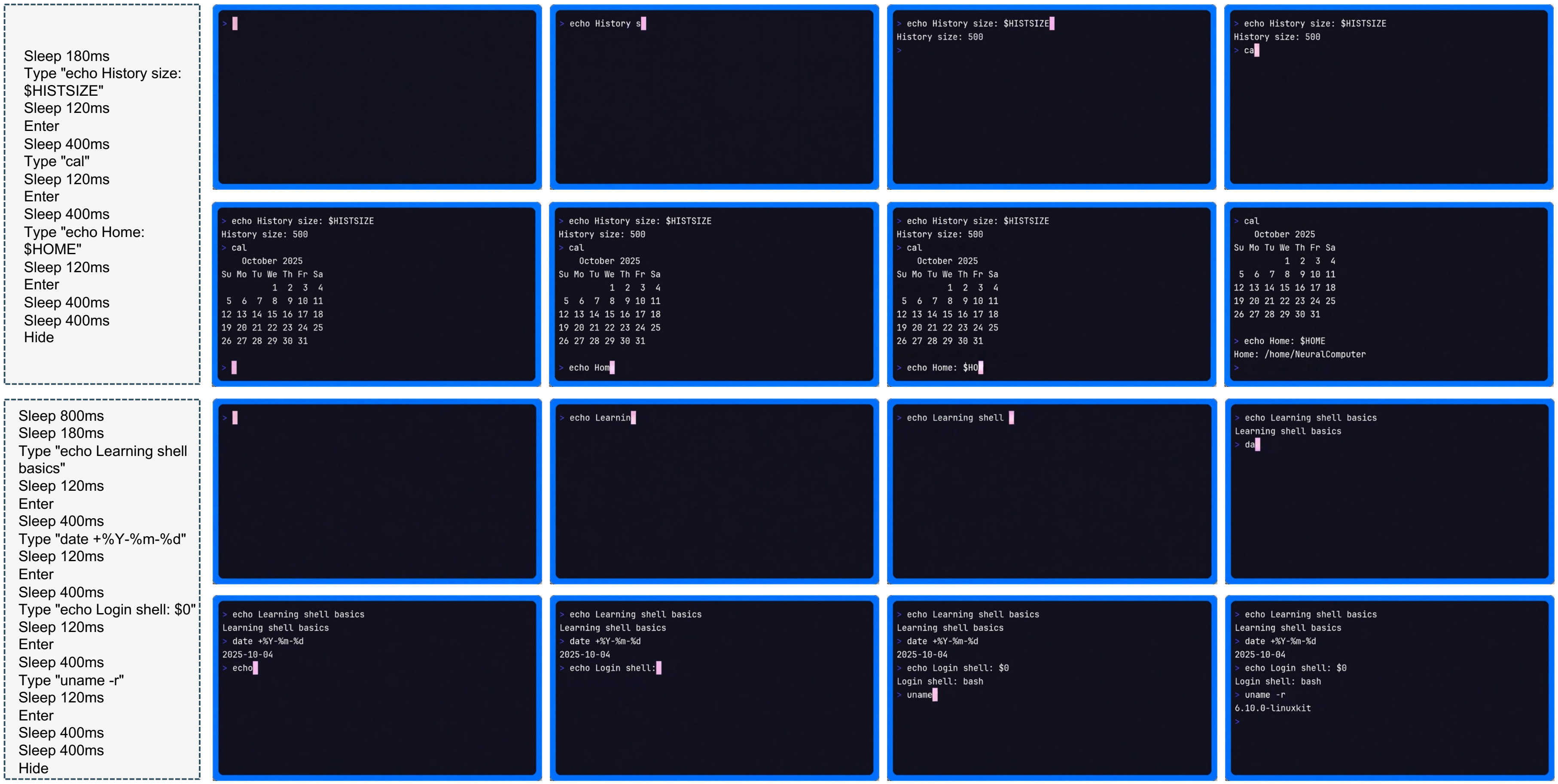


Figure 19 CLIGen (Clean) REPL visualization samples (D).

Back to Main Thumbnails

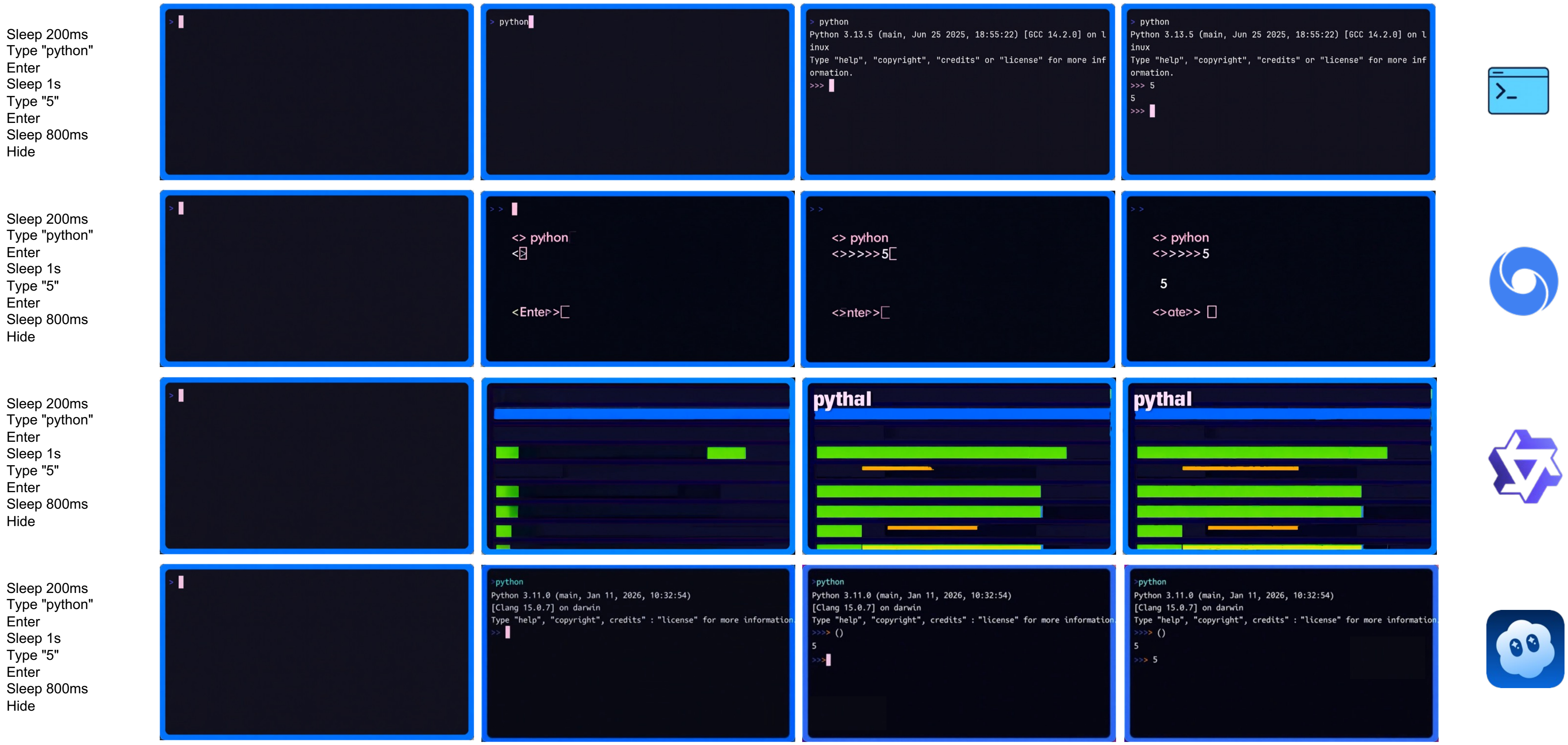


Figure 20 CLIGen (Clean) math comparison samples (A).

Back to Main Thumbnails

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "10+15"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "10+15"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "10+15"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "10+15"
Enter
Sleep 800ms
Hide

Figure 21 CLIGen (Clean) math comparison samples (B).

Back to Main Thumbnails

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "40/1"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "40/1"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "40/1"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "40/1"
Enter
Sleep 800ms
Hide

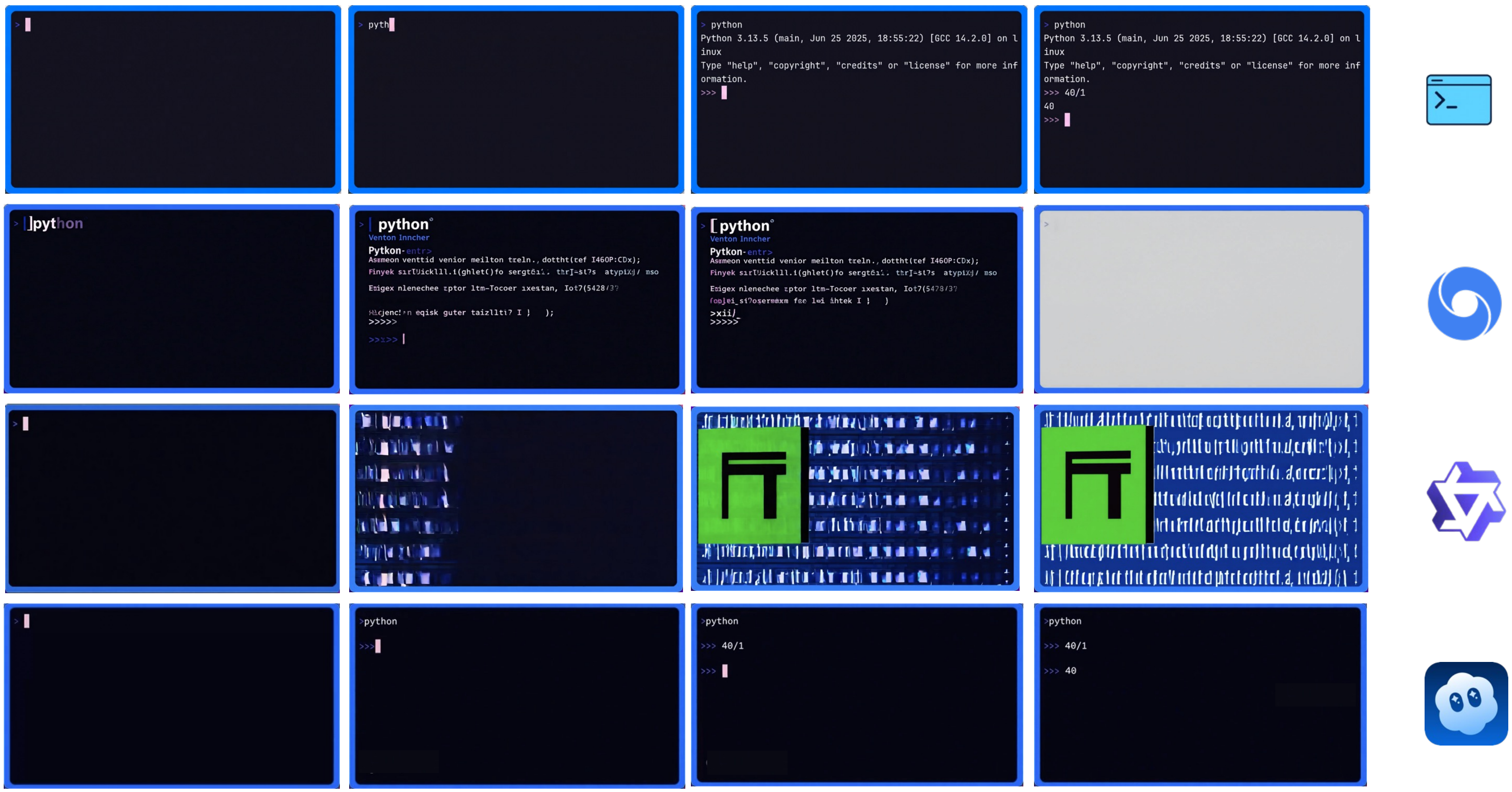


Figure 22 CLIGen (Clean) math comparison samples (C).

Back to Main Thumbnails

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "28-23"
Enter
Sleep 800ms
Hide
The answer is 5

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "28-23"
Enter
Sleep 800ms
Hide

Sleep 200ms
Type "python"
Enter
Sleep 1s
Type "28-23"
Enter
Sleep 800ms
Hide

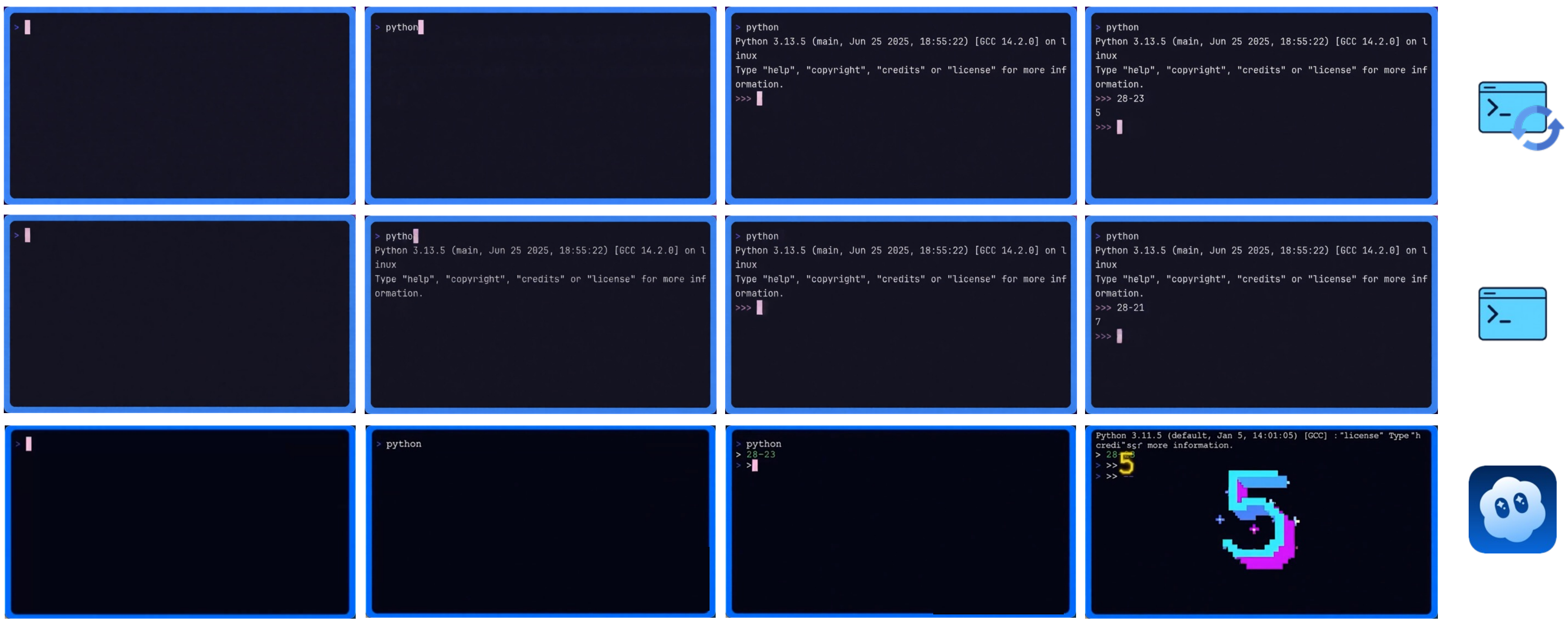


Figure 23 CLIGen (Clean) math reprompting samples (A).

Back to Main Thumbnails

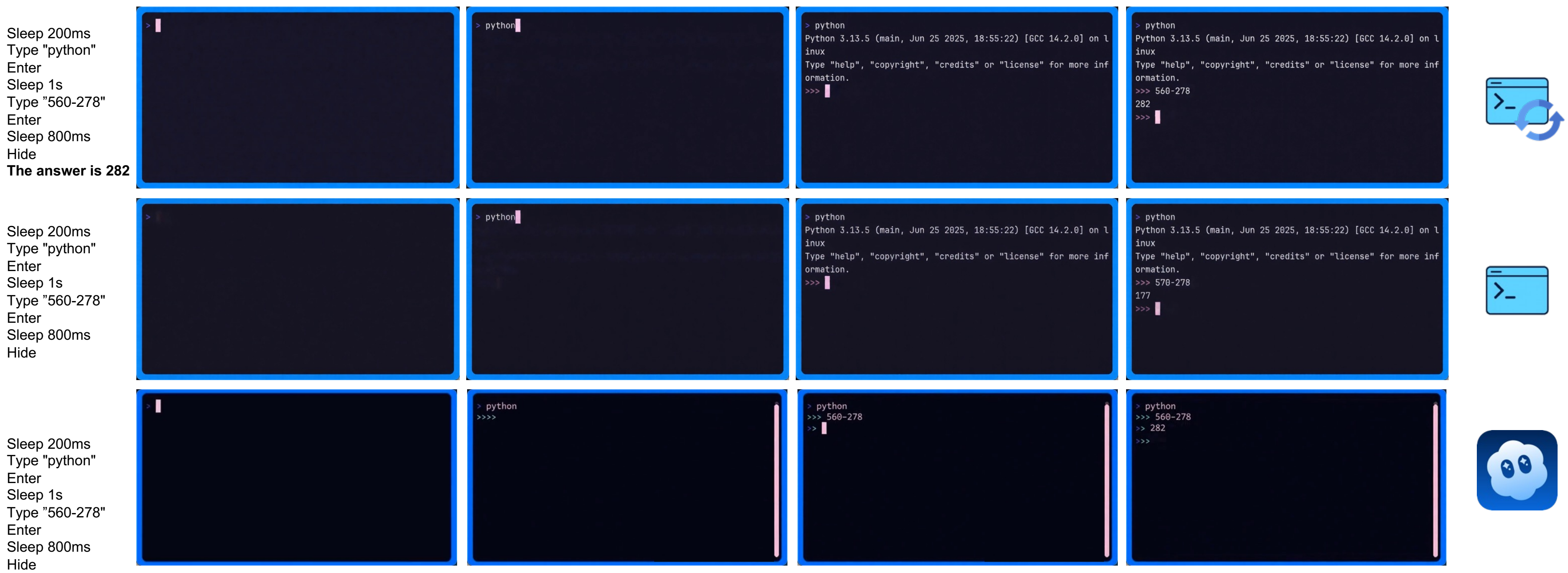


Figure 24 CLIGen (Clean) math reprompting samples (B).

Back to Main Thumbnails

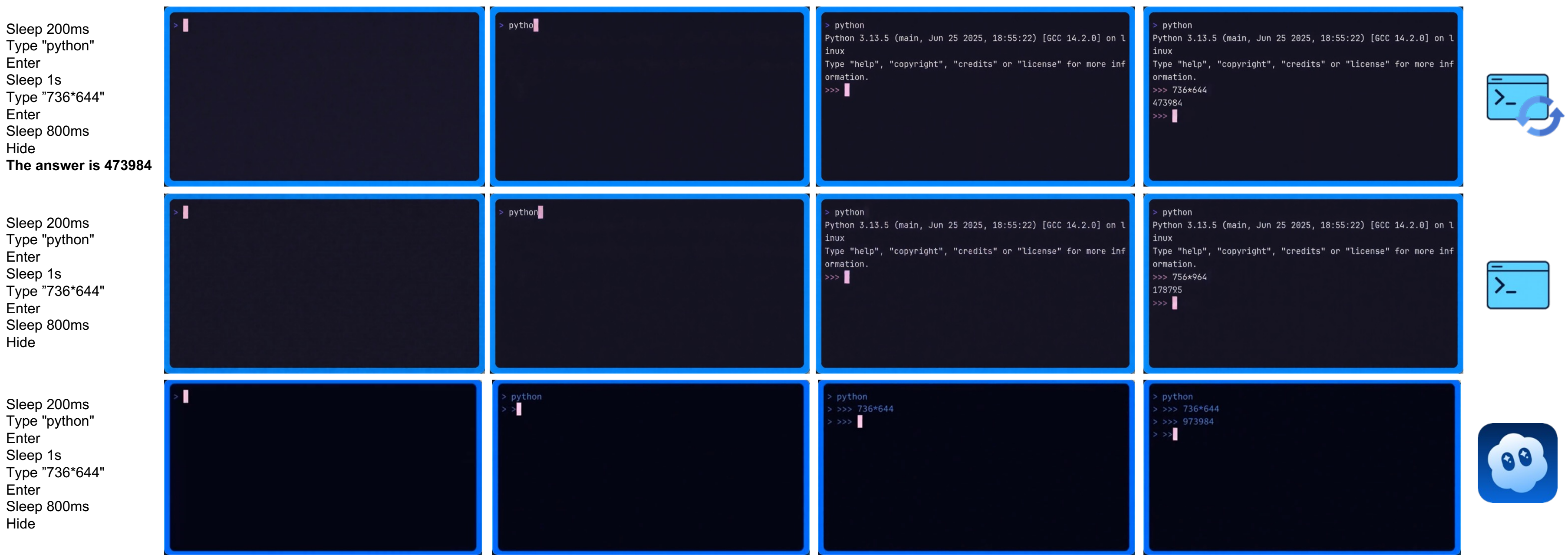


Figure 25 CLIGen (Clean) math reprompting samples (C).

Back to Main Thumbnails

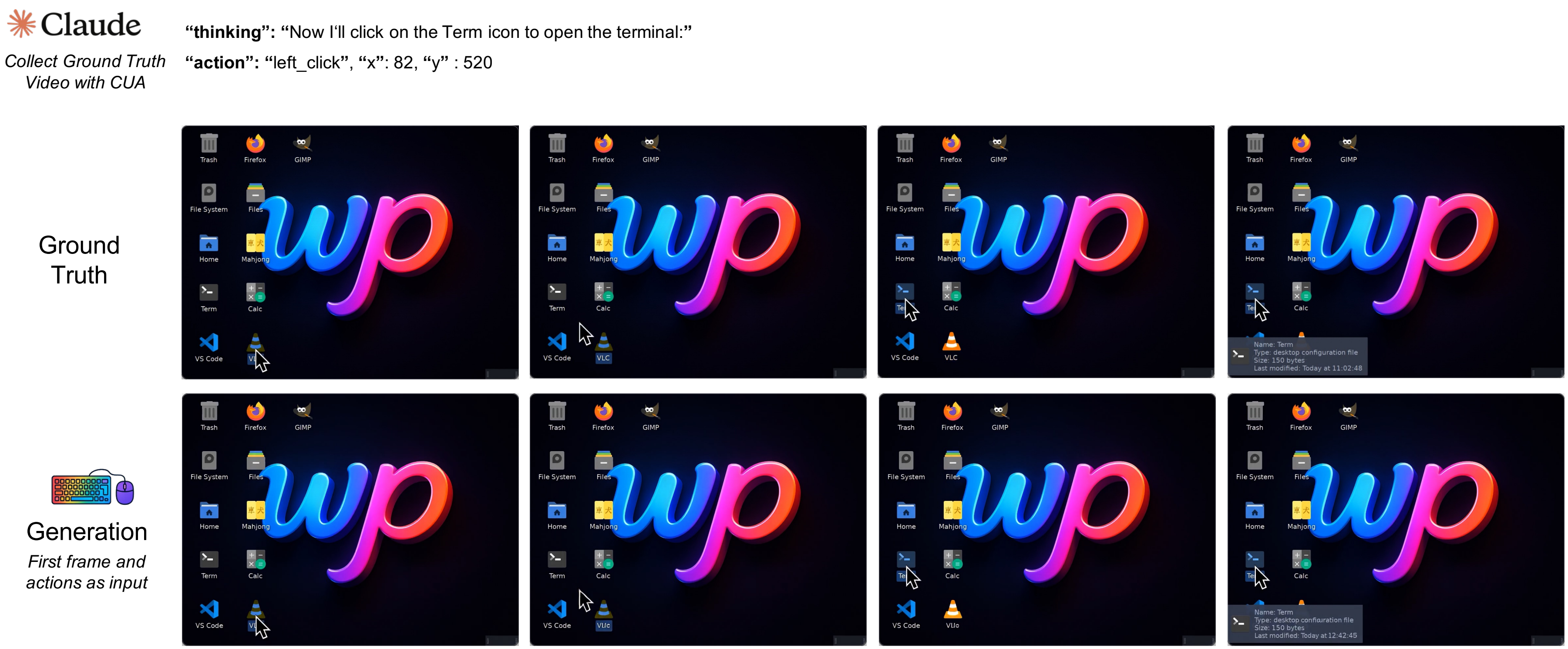


Figure 26 GUIWorld visualization sample (1).

Back to Main Thumbnails

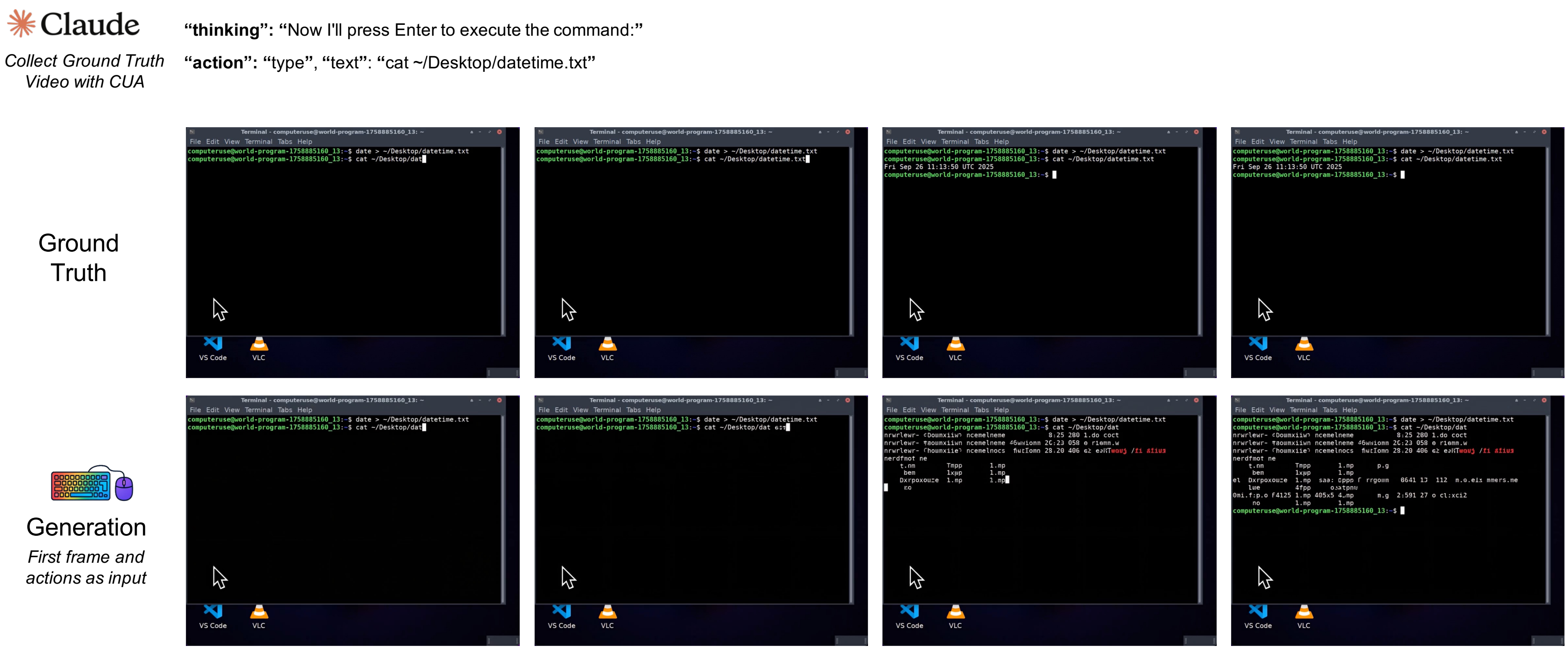


Figure 27 GUIWorld visualization sample (2).

Back to Main Thumbnails

Claude

Collect Ground Truth Video with CUA

"thinking": "I see Wikipedia appears in the suggestions. I'll click on it to go to Wikipedia:"

"action": "left_click", "x": 240, "y" : 150

Ground Truth

Generation

First frame and actions as input

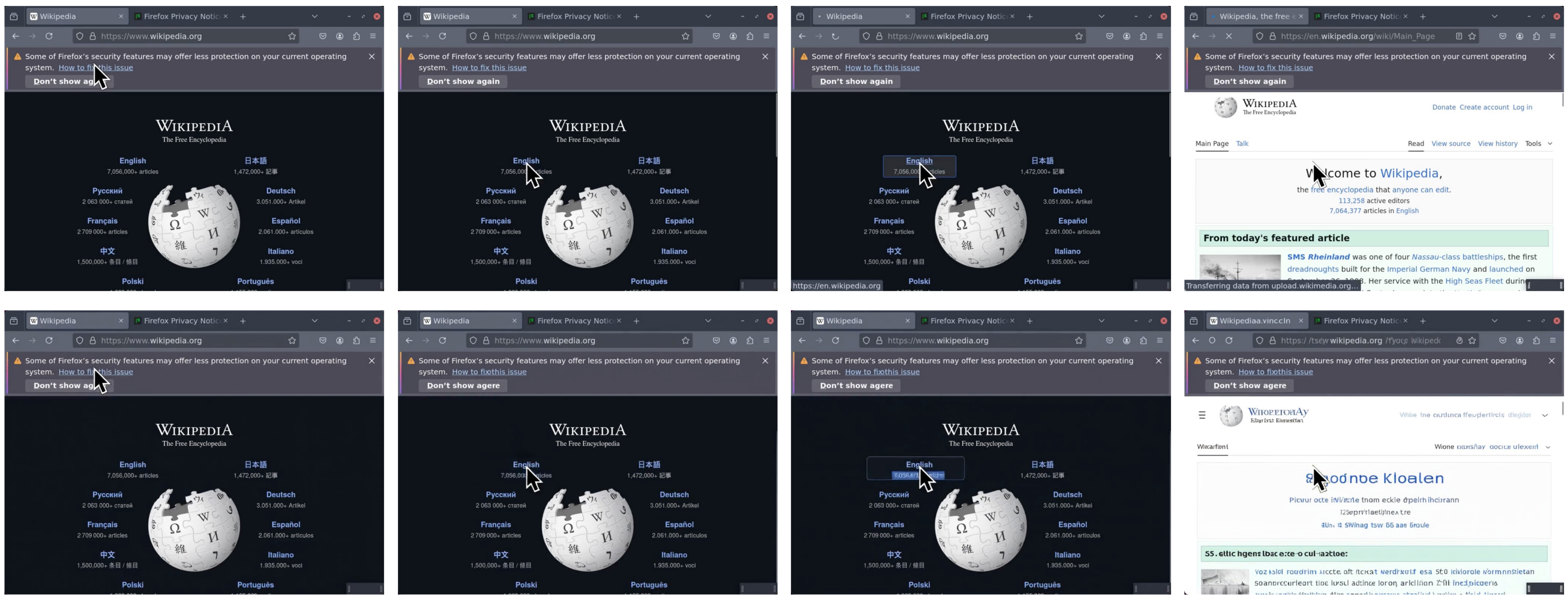


Figure 28 GUIWorld visualization sample (3).

Back to Main Thumbnails

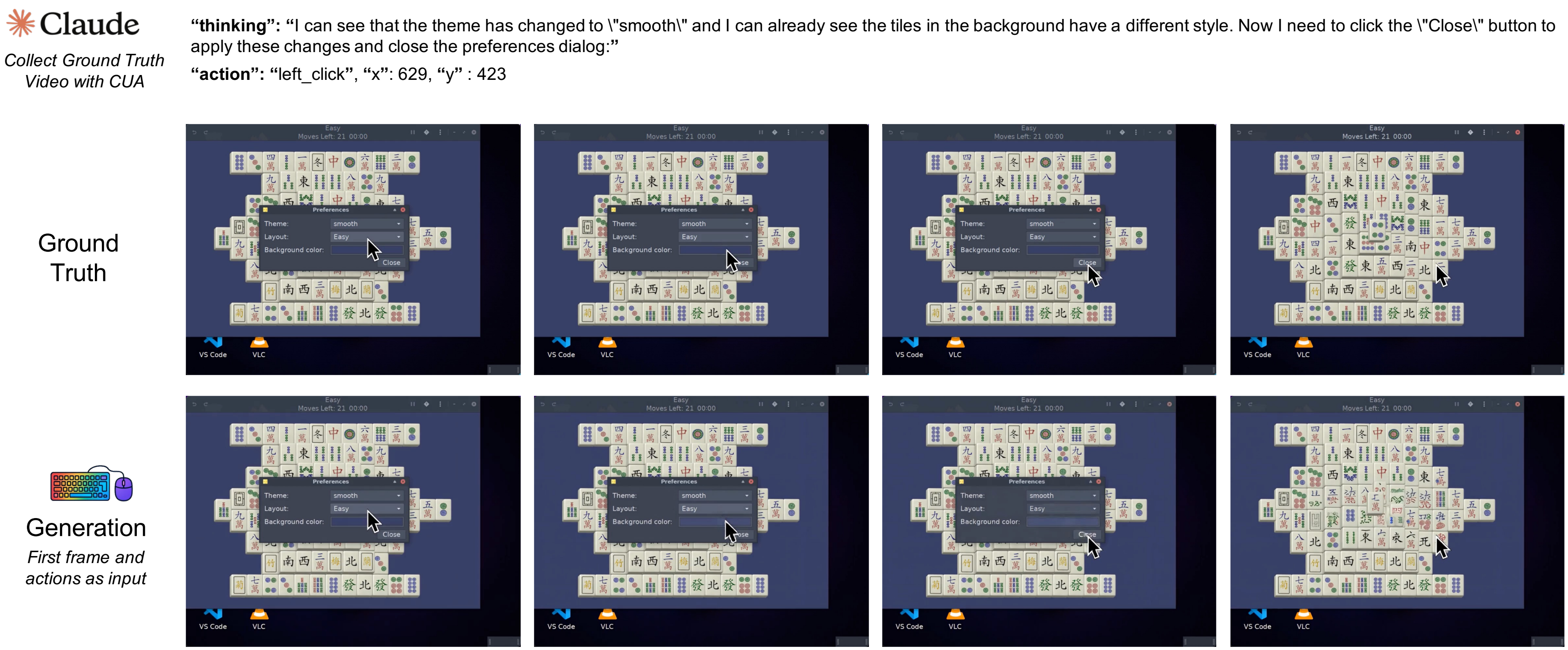


Figure 29 GUIWorld visualization sample (4).

Back to Main Thumbnails

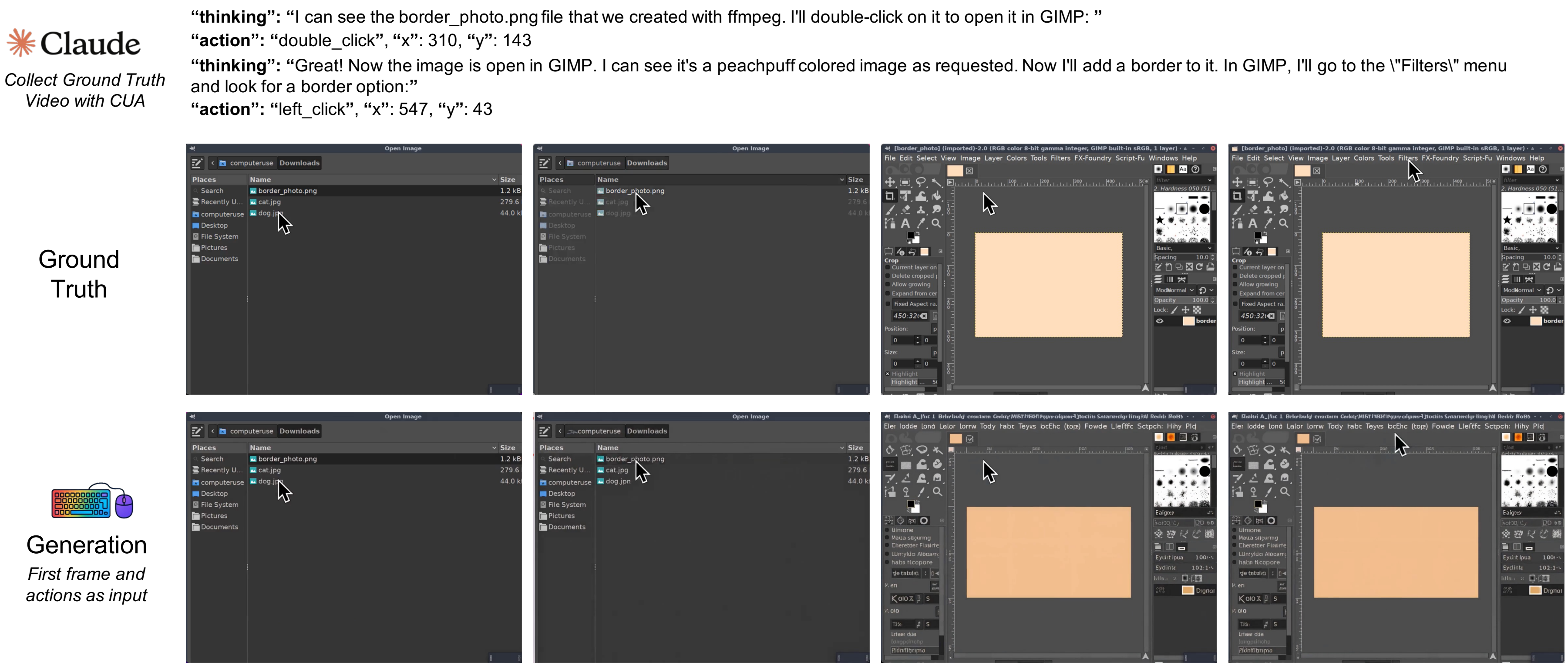


Figure 30 GUIWorld visualization sample (5).

Back to Main Thumbnails

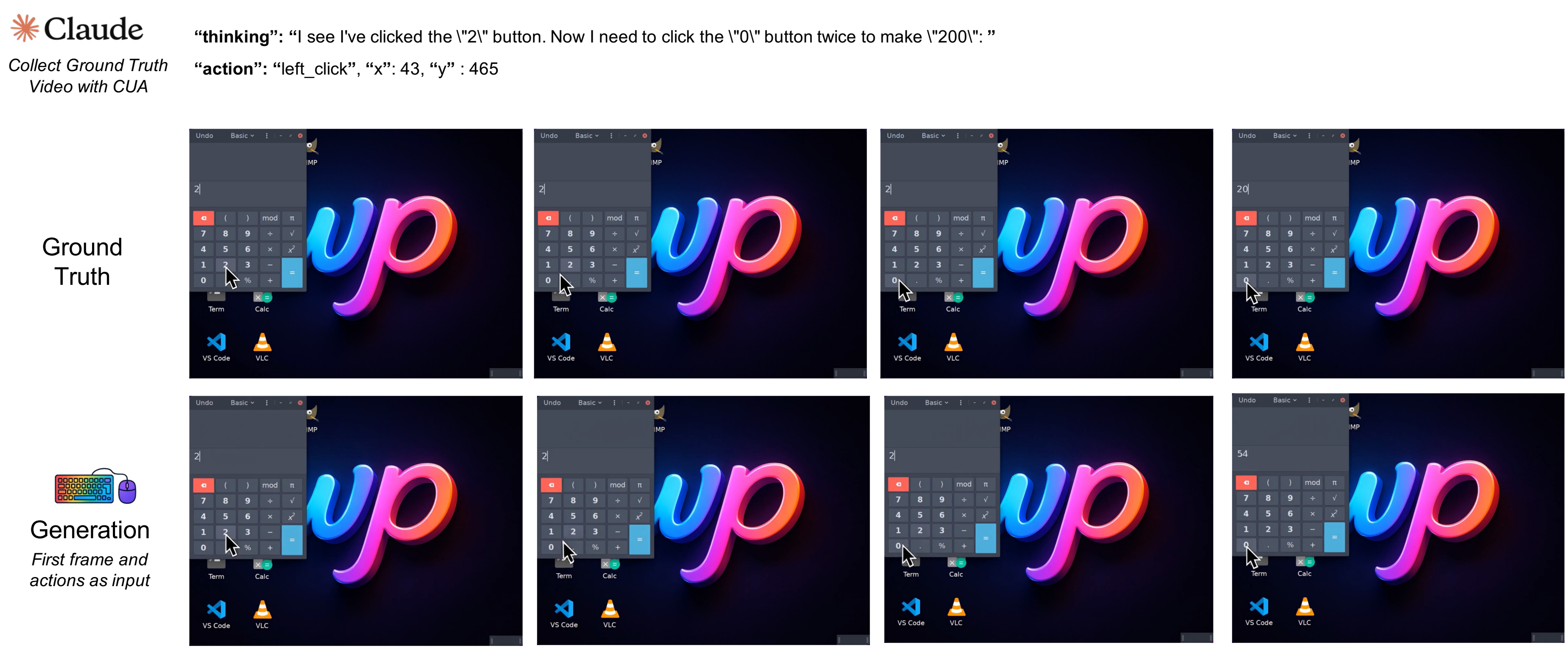


Figure 31 GUIWorld visualization sample (6).

Back to Main Thumbnails

Claude

Collect Ground Truth Video with CUA

"thinking": "Perfect! GIMP is now open. Let's import an image from the Downloads folder. I'll click on the File menu and select Open:"

"action": "left_click", "x": 43, "y" : 465

Ground Truth

Generation

First frame and actions as input

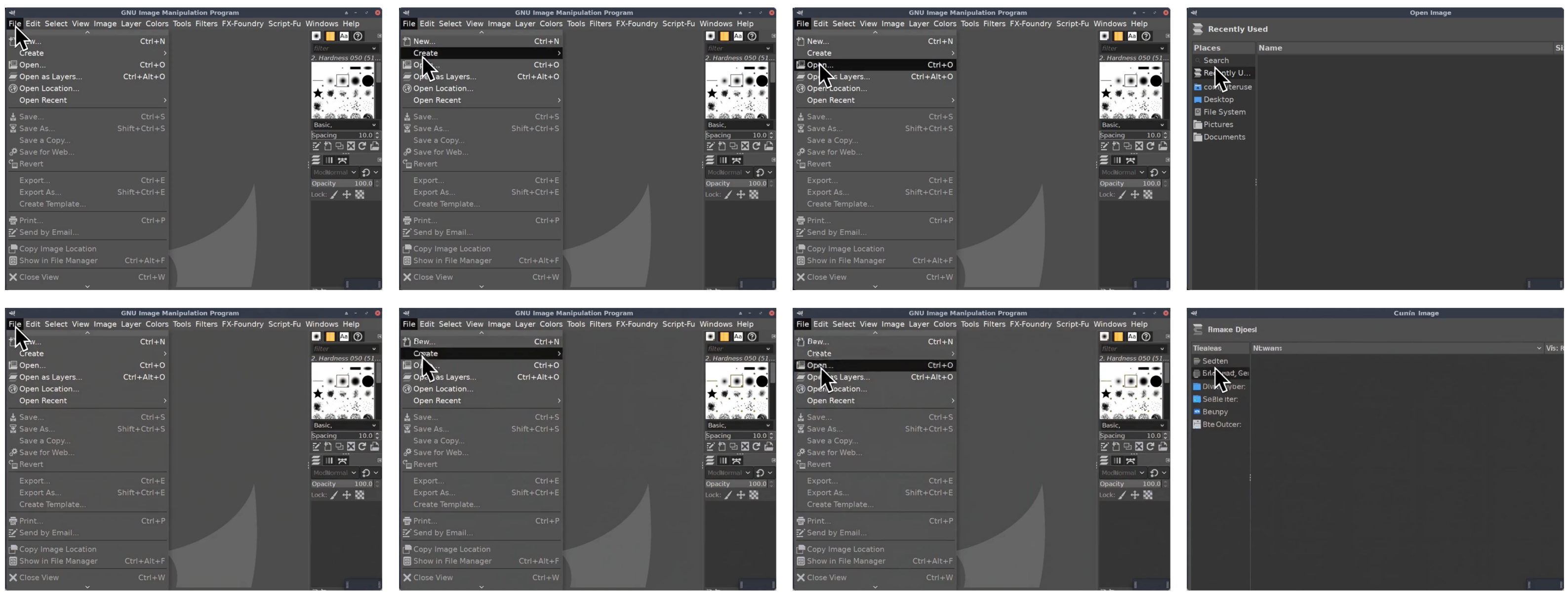


Figure 32 GUIWorld visualization sample (7).

Back to Main Thumbnails

Claude

Collect Ground Truth Video with CUA

"thinking": "Excellent! Nautilus is open and showing the home directory. Now I need to navigate to the Downloads folder. I can see the Downloads folder in the current view, so I'll double-click on it:"
"action": "double_click", "x": 248, "y" : 408

Ground Truth

Generation

First frame and actions as input

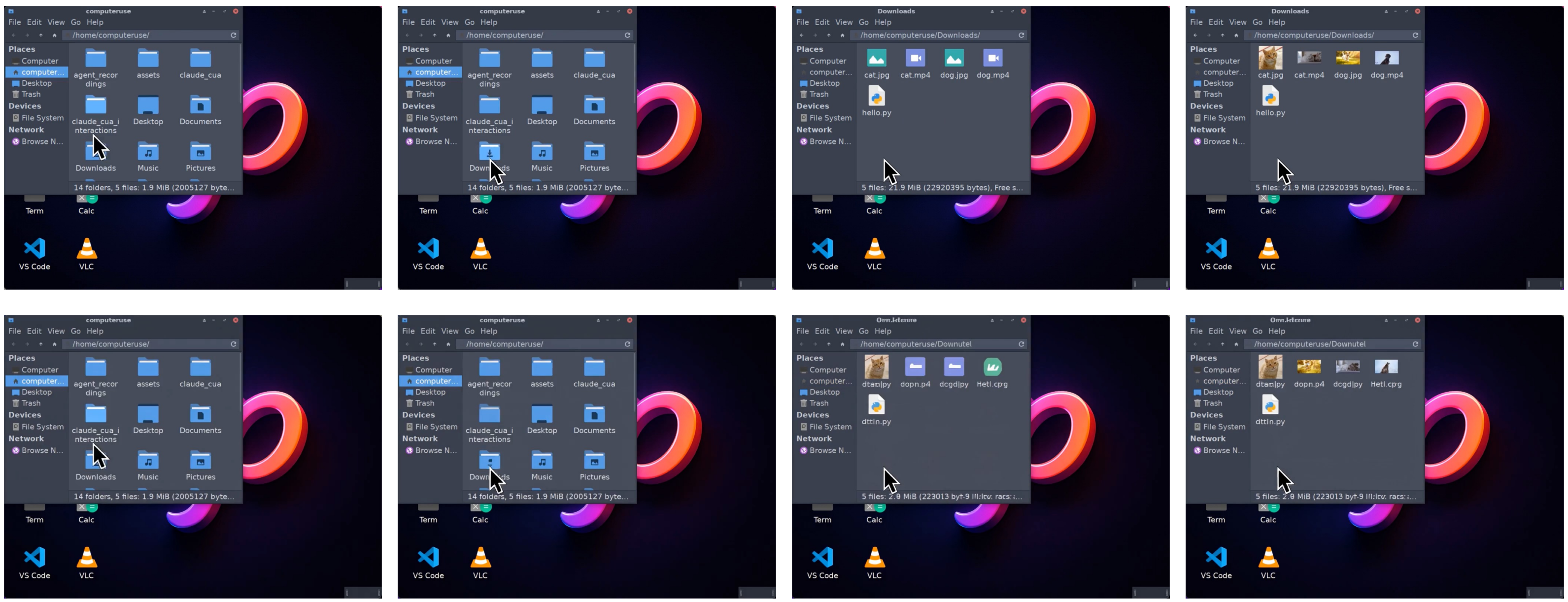


Figure 33 GUIWorld visualization sample (8).

Back to Main Thumbnails

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

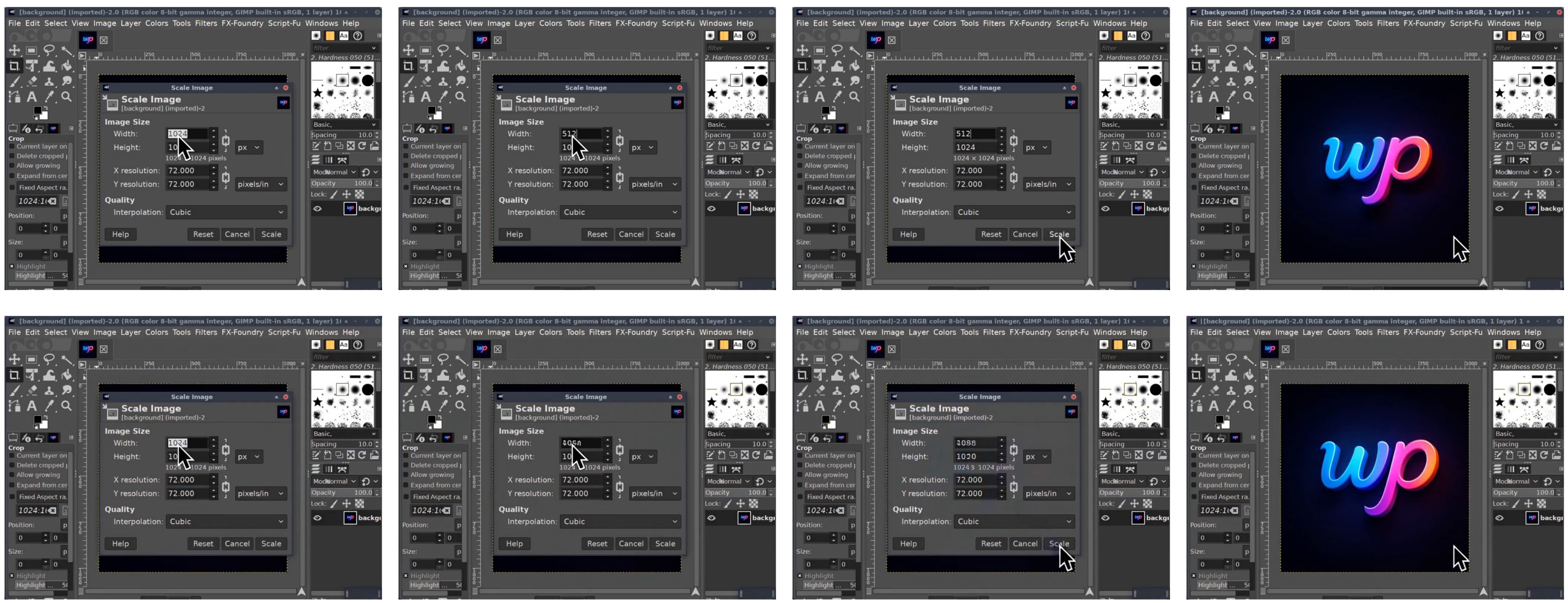


Figure 34 GUIWorld visualization sample (9).

Back to Main Thumbnails

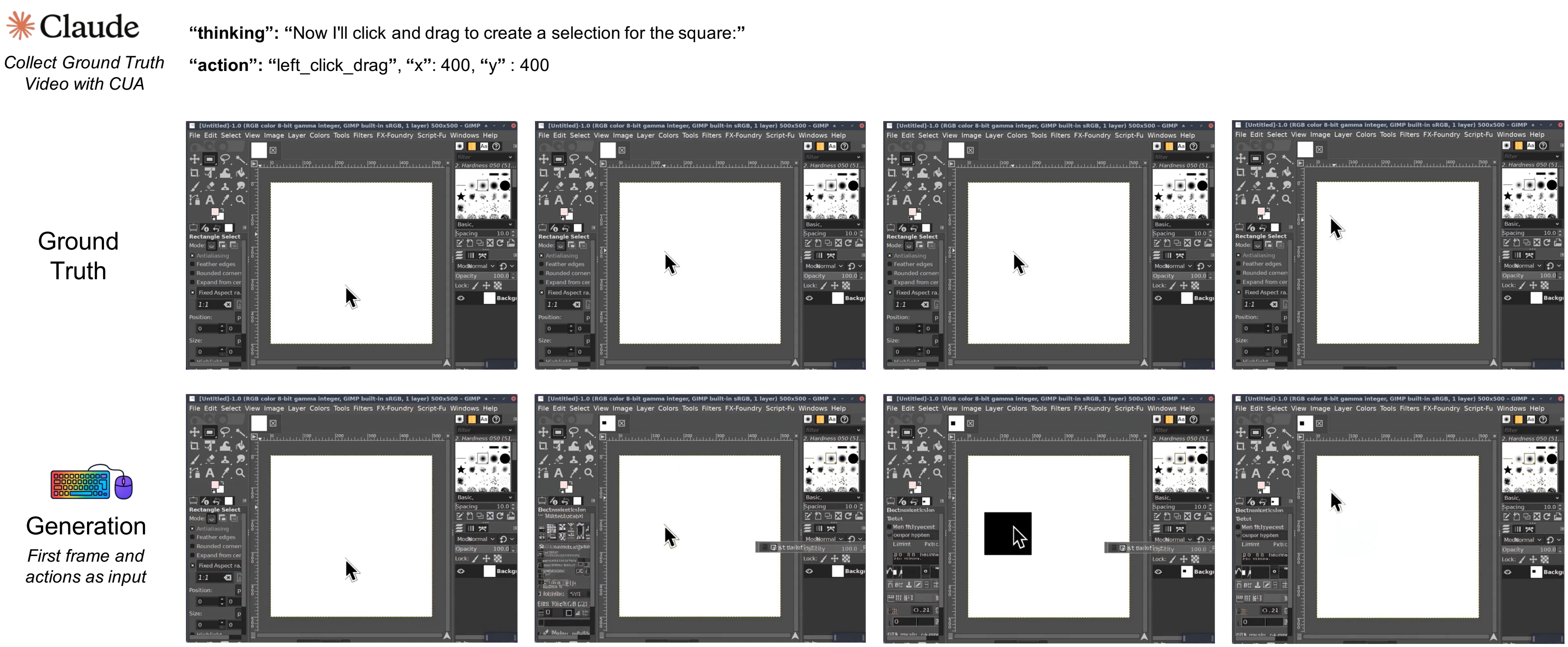


Figure 35 GUIWorld visualization sample (10).

Back to Main Thumbnails

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

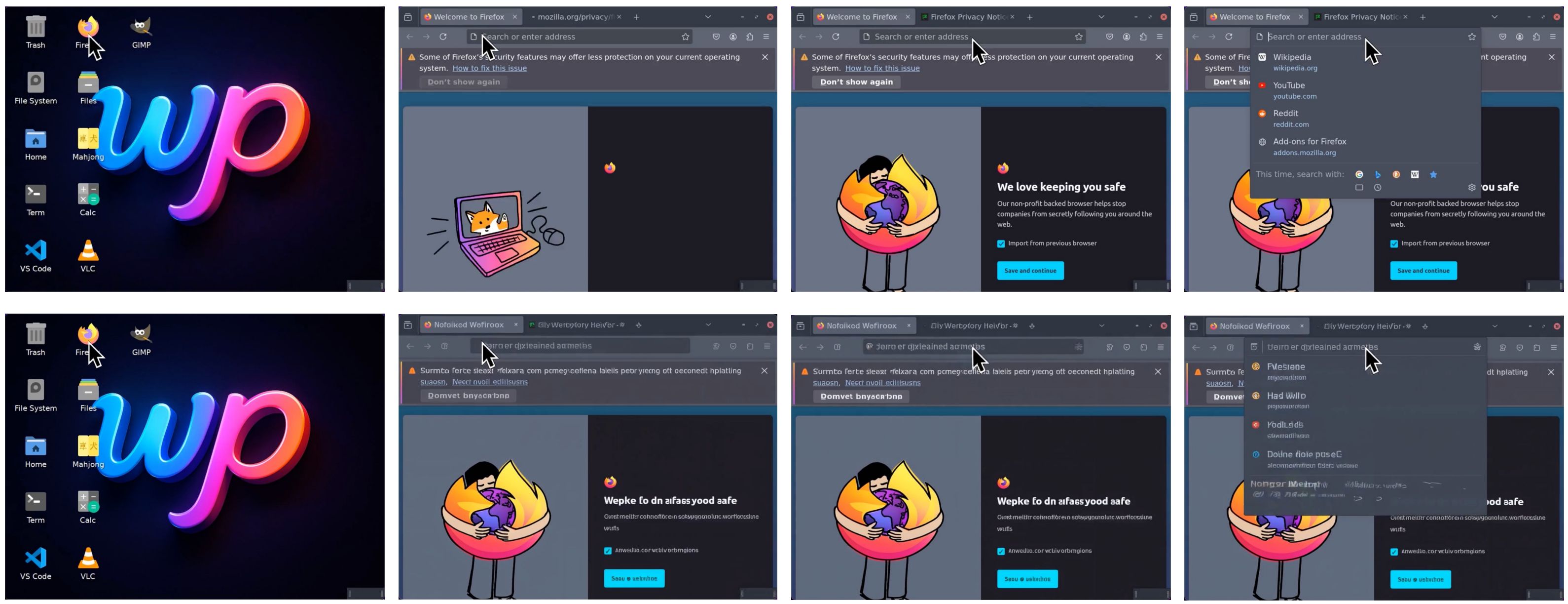


Figure 36 GUIWorld visualization sample (11).

Back to Main Thumbnails

Claude

Collect Ground Truth Video with CUA

"thinking": ""Now I'll type in 768 for the height:"
"action": "type", "text": "512"
"thinking": "Now I'll click the OK button to create the new image:"
"action": "left_click", "x": 763, "y": 514

Ground Truth

Generation

First frame and actions as input

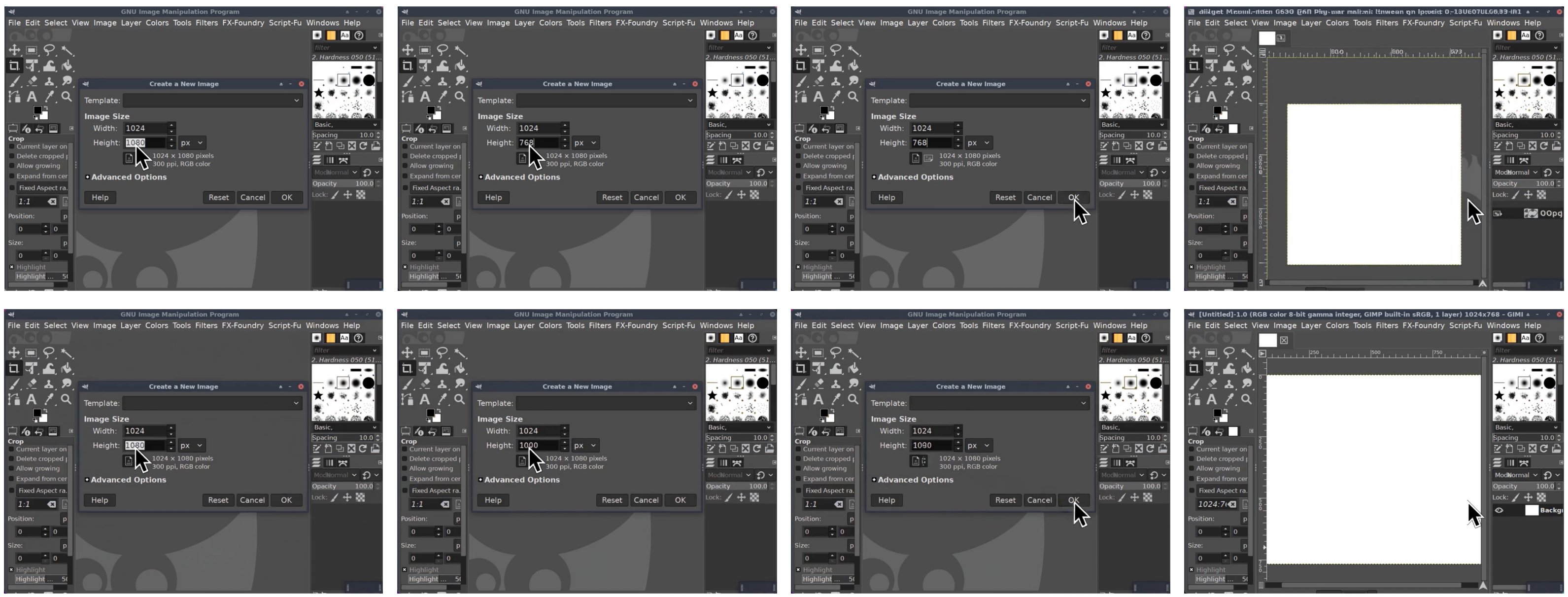


Figure 37 GUIWorld visualization sample (12).

Back to Main Thumbnails

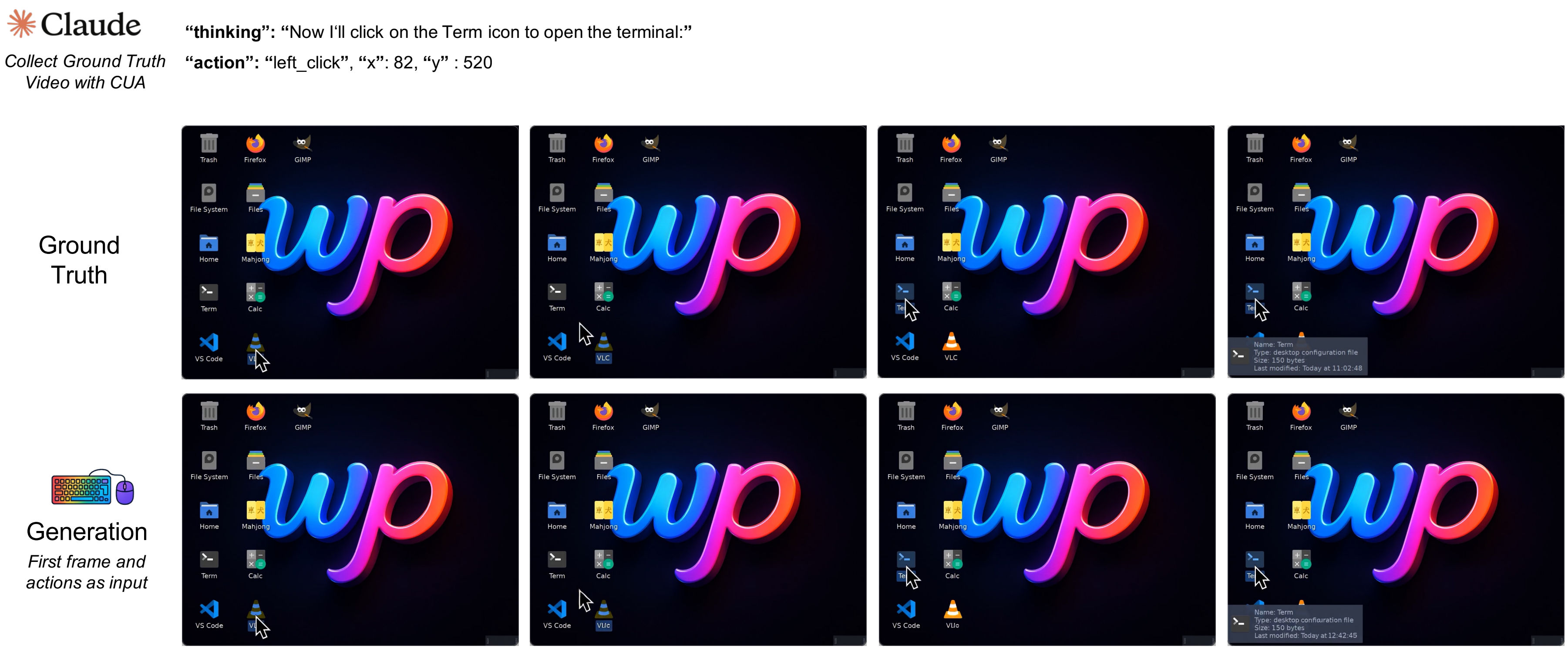


Figure 38 GUIWorld visualization sample (13).

Back to Main Thumbnails

Claude

Collect Ground Truth Video with CUA

"thinking": "I see the page has pagination at the bottom. Let me continue navigating to find the Advanced section. I'll click on page 2 to see more tutorials."

"action": "left_click", "x": 448, "y" : 539

Ground Truth

Generation

First frame and actions as input

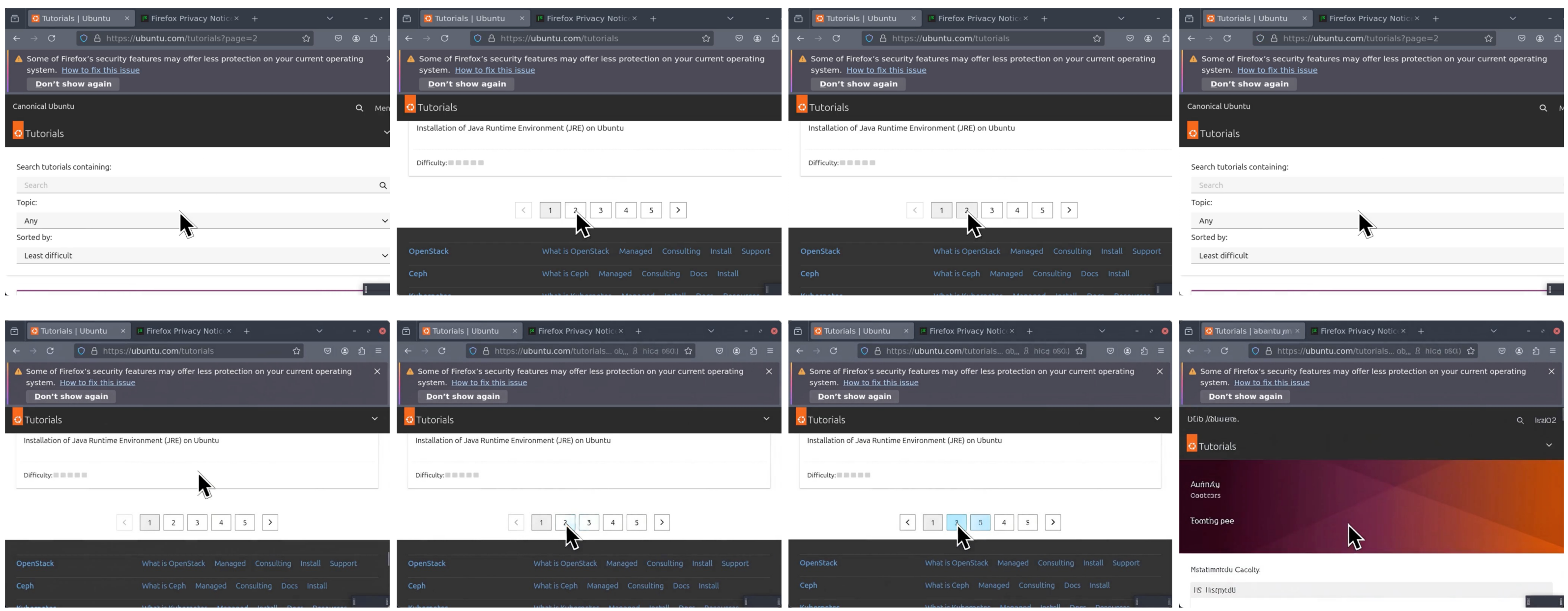


Figure 39 GUIWorld visualization sample (14).