Multimodal tool use & modular agents

SEMINAL

ViperGPT

Surís, Menon & Vondrick · ICCV 2023

FRONTIER

CoAct-1

Song et al. · ICLR 2026

Presented by Aadi Rane, Changda Ma

Georgia Tech

Presentation overview

How can a language model use code to do more than one model can alone?

SEMINAL PAPER · 2023

ViperGPT

An LLM writes a Python program that calls vision models

Query + image → answer

FRONTIER PAPER · 2026

CoAct-1

A manager picks code or clicks for each subtask, then checks

Goal → finished task

Same principle: *an LLM writes executable steps that call specialist tools*

ViperGPT

Surís, Menon & Vondrick · ICCV 2023 · Columbia University

Visual Inference via Python Execution for Reasoning

Seminal paper: code that composes vision models to answer a query

Georgia Tech

Outline

- *Problem statement and related work*
- *Approach: code generation, the API, code execution*
- *Worked examples: grounding, compositional QA, external knowledge, video*
- *Results and the intervention study*
- *Limitations and societal implications*
- *Discussion, then CoAct-1: what persisted and what changed*

Georgia Tech

Problem: one question, several reasoning steps



Query: *How many muffins can each kid have for it to be fair?*

Find kids and muffins → Count how many of each → Determine "fair" = division

Why not one model? End-to-end VLMs mix perception and reasoning in one pass: hard to inspect, and new tasks need retraining.

ViperGPT, Figure 1 and Section 1.

Related work: where does ViperGPT sit?

1

Modular vision

Compose modules per question

Neural Module Networks build a network from modules for each question, but the modules are trained jointly on task data.

Examples Andreas et al. 2016; Johnson et al. 2017

2

End-to-end VLMs

One model does it all

Image and question in, answer out. Strong zero-shot, but no explicit steps to inspect or reuse.

Examples BLIP-2, Flamingo

3

LLMs that write code or call tools

Let the LLM plan in code

Language models write programs or call tools to solve text tasks. VisProg (concurrent) also composes vision modules with generated programs.

Examples PAL, Program of Thoughts, Toolformer; VisProg

ViperGPT = composition like (1) + pretrained models from (2) + an LLM that writes Python, as in (3)

What's new: general Python written by a code LLM, calling only pretrained models. No training.

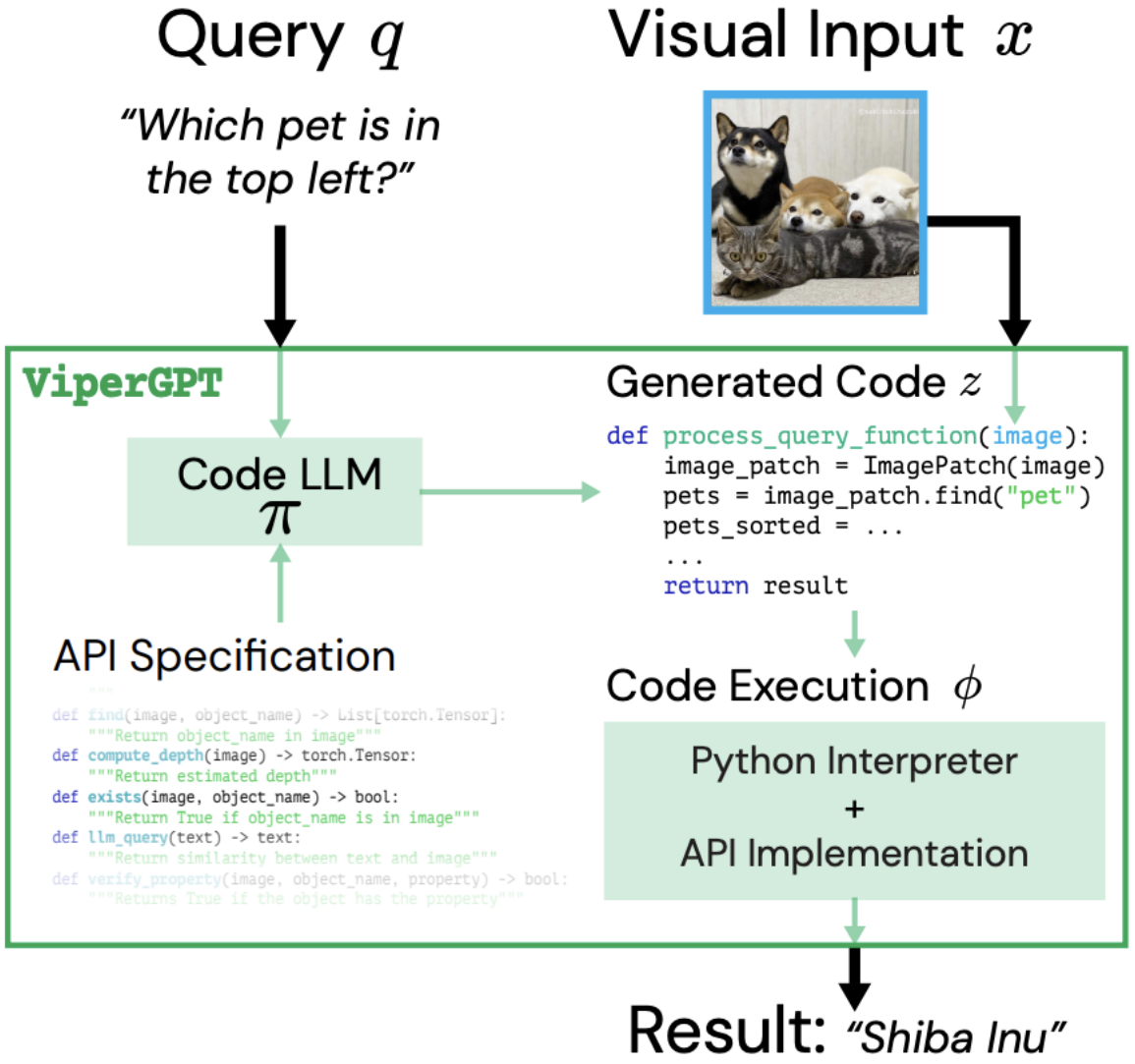
ViperGPT, Sections 1–2. Examples are works the paper cites; the summary line is presenter analysis.

Georgia Tech

Approach: query → program → answer

ViperGPT turns a visual query into a **programming task**.

Given an API of pretrained vision models, a code LLM writes a Python program. Running the program returns the answer.



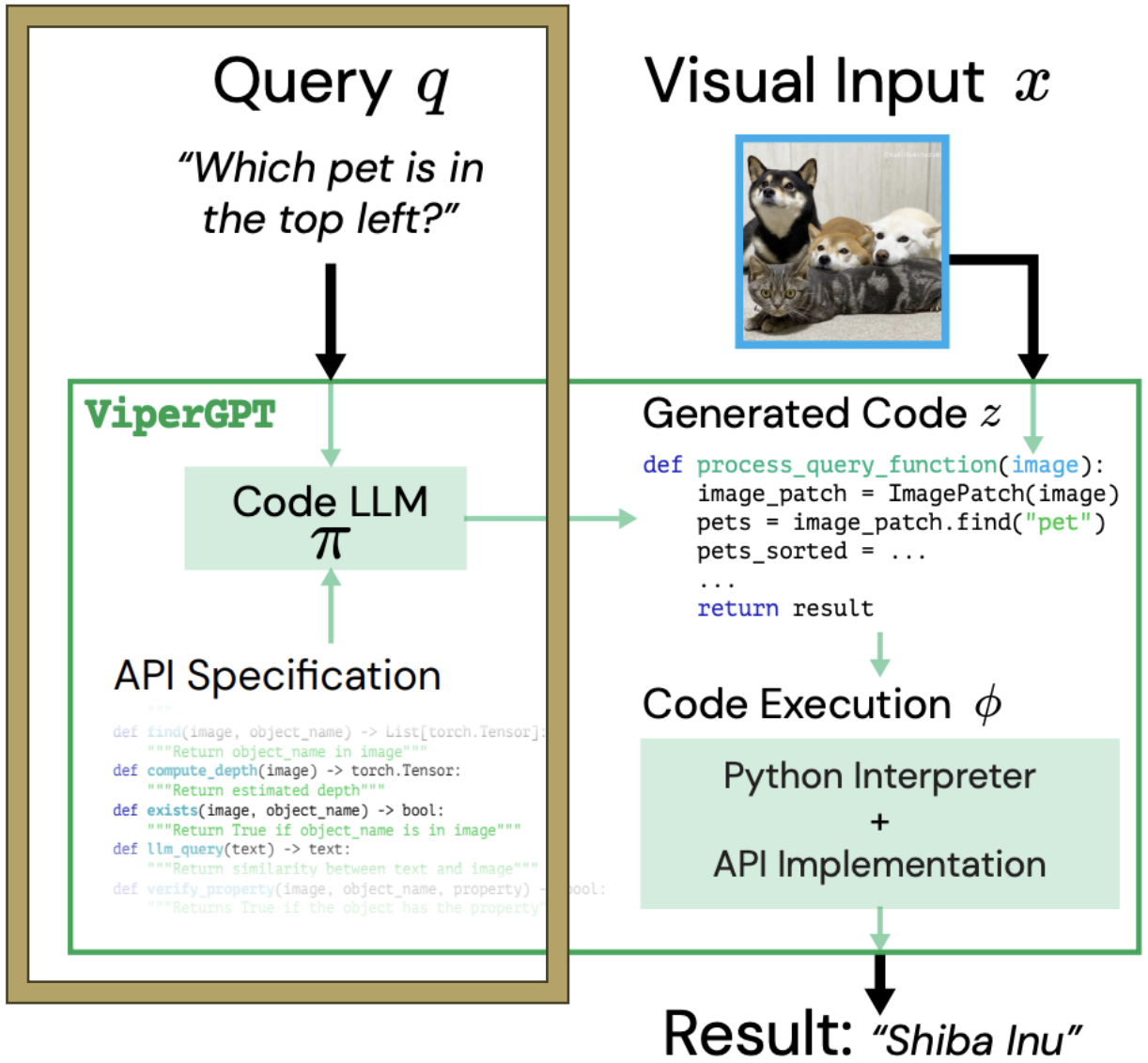
ViperGPT, Figure 2.

Georgia Tech

Approach: query → program → answer

ViperGPT turns a visual query into a **programming task**.

Given an API of pretrained vision models, a code LLM writes a Python program. Running the program returns the answer.



ViperGPT, Figure 2.

Georgia Tech

Step 1: Codex writes the program

API specification

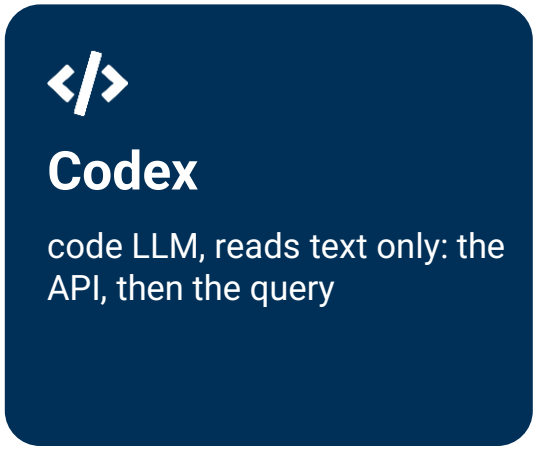
signatures and docstrings, no implementation

```
find(object_name)
exists(object_name)
verify_property(name, prop)
simple_query(question)
…
```

Query

“Which pet is in the top left?”






```
def execute_command(image):
  image_patch = ImagePatch(image)
  pets = image_patch.find("pet")
  pets_sorted = …
  …
  return result
```

generated Python program

Written blind: Codex never sees the image. It only reads the query and the API, so the program is a plan made before looking.

ViperGPT, Section 3 (“The input to Codex … is given the specification for the API, including the function signatures and docstrings”), Figure 2.

Georgia Tech

The API: pretrained models behind simple functions

`find()`

GLIP

Image patches that contain the object

`exists()`

GLIP

Is the object in the image?

`verify_property()`

X-VLM

Does the object have this property?

`best_image_match()`

X-VLM

Which patch best matches a description?

`best_text_match()`

X-VLM

Which description best matches the patch?

`compute_depth()`

MiDaS

Median depth of the patch

`distance()`

plain Python

Pixel distance between two patches

`simple_query()`

BLIP-2

Answers a simple question about the image

`llm_query()`

GPT-3

Answers from world knowledge; sees text only

8 of 9

functions call a pretrained model.

None is trained for these tasks; `distance()` is plain Python.

Swap a model behind a function, and every program that calls it improves.

ViperGPT, Section 3 and Appendix B. The takeaway is presenter analysis of the modular design.

Georgia Tech

Step 2: run the program

```
pets = image_patch.find("pet")
pets_sorted = …
return result
```

the generated program

API implementation

GLIP, X-VLM, MiDaS, BLIP-2, GPT-3, run on the image

Python interpreter

Runs the program line by line. Each API call runs a pretrained model; plain Python does the rest.

Result

“Shiba Inu”

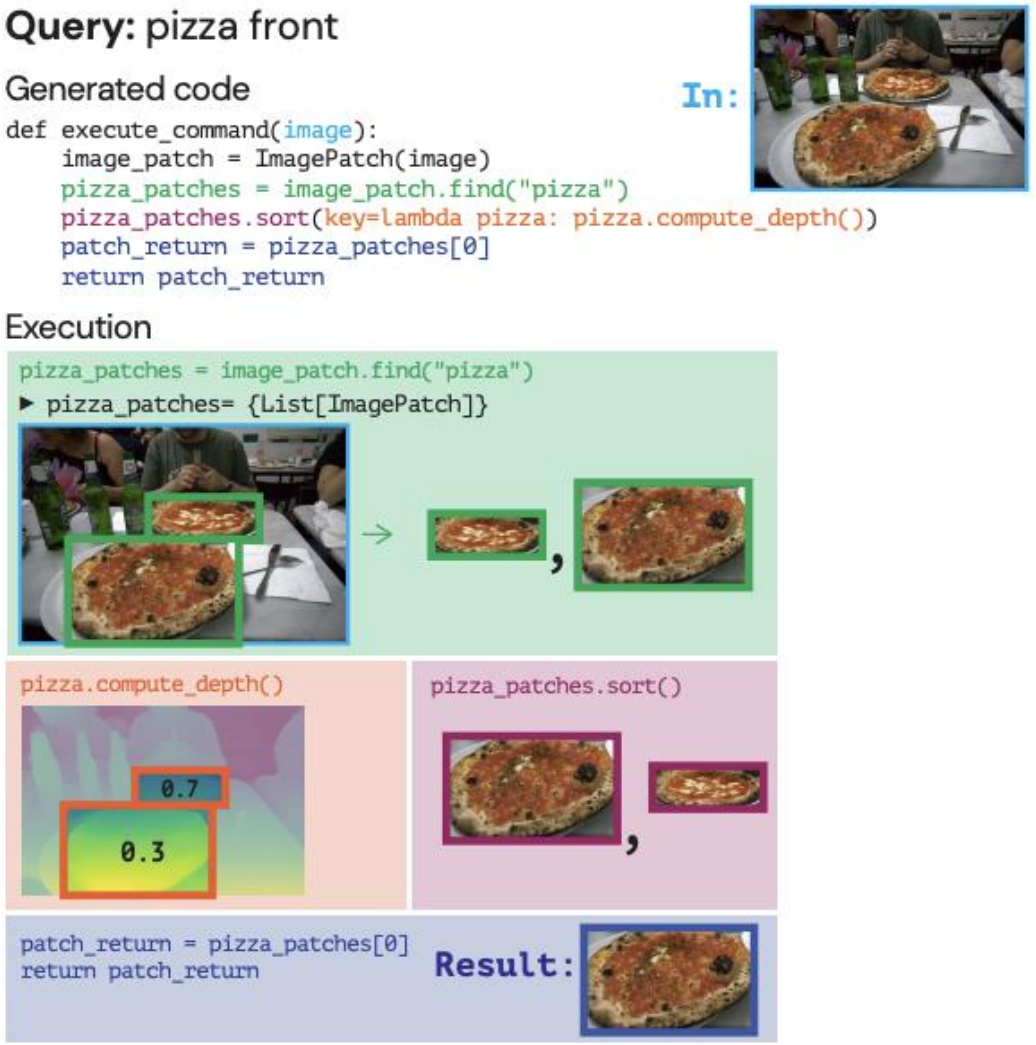
Inspectable: every intermediate value can be printed and checked.

Runs once: no check or retry, so a wrong module output flows into the answer.

ViperGPT, Sections 1 and 3, Figure 2. “Runs once” summarizes the method; the paper describes no retries.

Georgia Tech

Worked example: visual grounding (RefCOCO)



Query: *"pizza front"*

1. `find("pizza")` GLIP returns every pizza patch
2. `compute_depth()` MiDaS estimates each patch's depth
3. `sort(…)` plain Python orders them, nearest first
4. `[0]` return the first patch: the front pizza

"Front" becomes code: sort by depth, take the first. Two models, three lines.

ViperGPT, Figure 3.

Georgia Tech

Worked example: compositional QA (GQA)

Query: Are there water bottles to the right of the bookcase that is made of wood?

In:



Generated code

```
def execute_command(image):
    image_patch = ImagePatch(image)
    bookcase_patches = image_patch.find("bookcase")
    for bookcase_patch in bookcase_patches:
        is_wood = bookcase_patch.verify_property("bookcase", "wood")
        if is_wood:
            water_bottle_patches = image_patch.find("water bottle")
            for water_bottle_patch in water_bottle_patches:
                if water_bottle_patch.horizontal_center > \
                   bookcase_patch.horizontal_center:
                    return "yes"
            return "no"
    return "no"
```

Execution

```
bookcase_patches= image_patch.
        find("bookcase")
▶ bookcase_patches[0] = {ImagePatch}
```



```
▶ bookcase_patches[0].
      horizontal_center = {float} 239.0
...verify_property("bookcase","wood")
▶ is_wood = {bool} True
```

```
water_bottle_patches = image_patch.
        find("water bottle")
▶ water_bottle_patches[0]
    = {ImagePatches}
```



```
▶ water_bottle_patches[0].
      horizontal_center = {float} 608.5
▶ water_bottle_patch.horizontal_center >
bookcase_patch.horizontal_center =
{bool} True
```

Result: "yes"

`find("bookcase")` GLIP → `verify_property(…, "wood")` X-VLM → `find("water bottle")` GLIP → `compare x-centers` Python → "yes" answer

Each model does one small job; a loop, an if and a comparison do the logic.

ViperGPT, Figure 4.

Worked example: external knowledge (OK-VQA)

Query: The real live version of this toy does what in the winter?

Generated code

```
def execute_command(image):
    image = ImagePatch(image)
    toy = image.simple_query("What is this toy?")
    result = llm_query("The real live version of
                {} does what in the winter?", toy)
    return result
```

In:



Execution

```
▶ toy = {str} "bear"

▶ guess = {str} "hibernate"
```

Result: “hibernate”

BLIP-2 result: “ski”

`simple_query` **BLIP-2**
sees the toy: “bear”

→

`llm_query` **GPT-3**
knows what real bears do: “hibernate”

BLIP-2 alone: *“ski”*
The paper calls this a **programmatic chain-of-thought**.

ViperGPT, Figure 5 (“Programmatic chain-of-thought with external knowledge for OK-VQA”).

Georgia Tech

Worked example: video question answering (NExT-QA)

Query: What did the boy do after he dropped the sparkles on the floor?

Generated code

```
def execute_command(video, question, possible_answers):
    video_segment = VideoSegment(video)
    drop_detected = False
    for i, frame in enumerate(video_segment.frame_iterator()):
        if frame.exists("boy") and frame.exists("sparkles") and \
                frame.simple_query("is the boy dropping the sparkles?") == "yes":
            drop_detected = True
            break
    if drop_detected:
        index_frame = i + 1
    else:
        index_frame = video_segment.num_frames // 2
    frame_of_interest = ImagePatch(video_segment, index_frame)
    boys = frame_of_interest.find("boy")
    if len(boys) == 0:
        boys = [frame_of_interest]
    boy = boys[0]
    caption = boy.simple_query("What is this?")
    info = {
        "Caption of frame after dropping the sparkles": caption,
    }
    answer = select_answer(info, question, possible_answers)
    return answer
```

Execution

In:

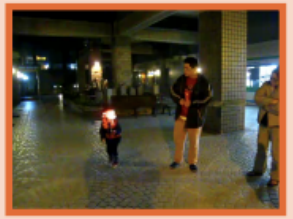


```
frame.exists("boy") and \
frame.exists("sparkles") and \
frame.simple_query("is the boy
   dropping the sparkles?") == "yes":
▶ frame = {ImagePatch}
```



```
▶ i= {int} 25
```

```
index_frame = i + 1
▶ index_frame = {int} 26
▶ frame_of_interest = {ImagePatch}
```



```
boys = frame_of_interest.find("boy")
▶ boy = {ImagePatch}
```



```
▶ caption = {str} "a child running
                   with fire in his hands"
▶ answer = {str} "pick it up"
```

Result: *"Pick it up"*

Time as a loop: plain Python walks the frames to find when the boy drops the sparklers, then looks at the next frame.

Multiple choice: NExT-QA gives the candidate answers; `select_answer()` (GPT-3) picks one from the gathered info.

ViperGPT, Figure 6 and Section 4 ("We evaluate using the NExT-QA multiple choice version").

Georgia Tech

Results: grounding and compositional QA

Table 1. **RefCOCO Results**. We report accuracy on the REC task and testA split. ZS=zero shot, Sup.=supervised.

		IoU (%) ↑	
		RefCOCO	RefCOCO+
Sup.	MDETR [53]	90.4	85.5
	OFA [53]	94.0	91.7
ZS	OWL-ViT [38]	30.3	29.4
	GLIP [31]	55.0	52.2
	ReCLIP [49]	58.6	60.5
	ViperGPT (ours)	**72.0**	**67.0**

Best zero-shot: 72.0 vs 58.6 (ReCLIP) and 30.3 (OWL-ViT). Supervised OFA still reaches 94.0.

Table 2. **GQA Results**. We report accuracy on the test-dev set.

		Accuracy (%) ↑
Sup.	LGCN [20]	55.8
	LXMERT [51]	60.0
	NSM [24]	63.0
	CRF [39]	72.1
ZS	BLIP-2 [30]	44.7
	ViperGPT (ours)	**48.1**

Best zero-shot: 48.1 vs 44.7 (BLIP-2). Supervised models reach 55.8 to 72.1.

Our takeaway: ViperGPT beats the very models it calls: +17 points over GLIP on RefCOCO, +3.4 over BLIP-2 on GQA. Composition adds real value, but without training it stays well below supervised models.

ViperGPT, Tables 1–2. The comparison with its own modules is presenter analysis.

Results: external knowledge and video

Table 3. **OK-VQA Results.**

		Accuracy (%) ↑
Sup.	TRiG [13]	50.5
	KAT [16]	54.4
	RA-VQA [32]	54.5
	REVIVE [33]	58.0
	PromptCap [21]	58.8
ZS	PNP-VQA [52]	35.9
	PICa [60]	43.3
	BLIP-2 [30]	45.9
	Flamingo [1]	50.6
	ViperGPT (ours)	**51.9**

Best zero-shot: 51.9 vs 50.6 (Flamingo), and above one supervised model (TRiG, 50.5).

Table 4. **NExT-QA Results**. Our method gets overall state-of-the-art results (including *supervised* models) on the hard split. "T" and "C" stand for "temporal" and "causal" questions, respectively.

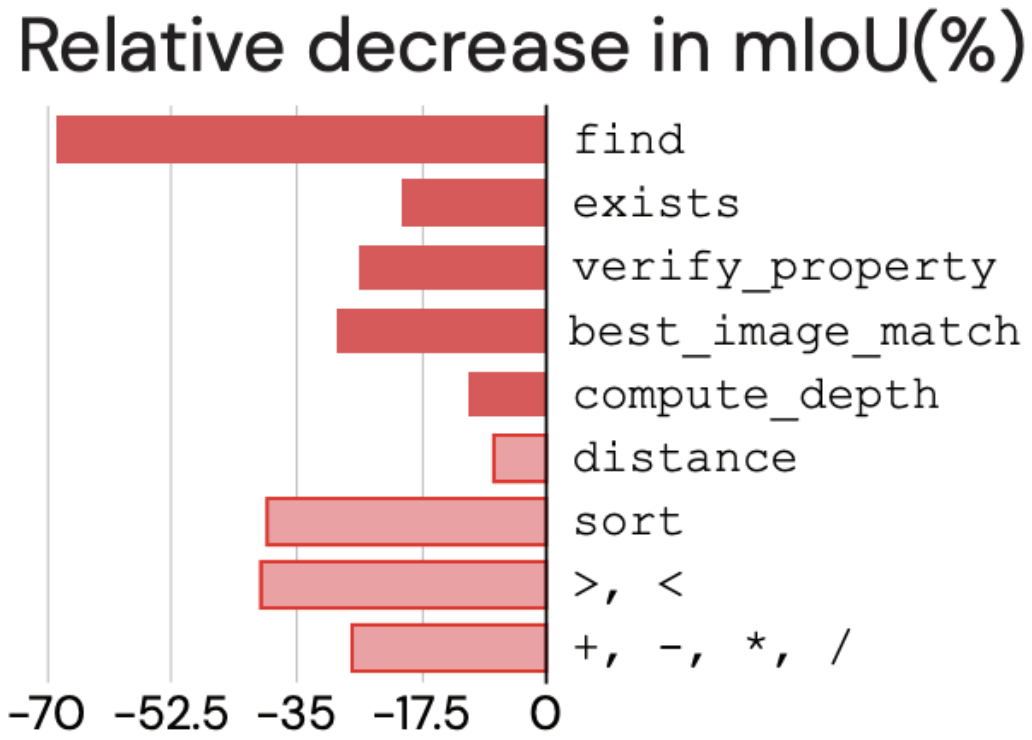
		Accuracy (%) ↑		
		Hard Split - T	Hard Split - C	Full Set
Sup.	ATP [7]	45.3	43.3	54.3
	VGT [58]	-	-	56.9
	HiTeA [61]	48.6	47.8	63.1
ZS	ViperGPT (ours)	**49.8**	**56.4**	60.0

No video training, yet above the best supervised model on the hard temporal and causal splits; below it on the full set (60.0 vs 63.1).

Our takeaway: the gains are largest where the program adds a step one model lacks: GPT-3 knowledge on OK-VQA (+6.0 over BLIP-2) and frame-by-frame search on hard causal video questions (+8.6 over HiTeA).

ViperGPT, Tables 3–4. The explanation of where the gains come from is presenter analysis.

Intervention: which parts of the program matter?



Protocol: disable one function at a time on RefCOCO and measure the relative drop in mIoU.

- `find()` matters most: it does most of the perception.
- **Python logic matters too:** `sort` and comparisons cost more than `exists()`, `verify_property()` or `best_image_match()`.
- *The paper: "a similar level of importance for perception modules and Python operations."*

Our takeaway: the reasoning really lives in the code: break the logic, and accuracy drops as much as when you break a vision model.

ViperGPT, Figure 7 and Section 4. Bar comparisons are read from the figure; the paper gives no exact values.

Georgia Tech

Why is this approach so powerful?

The authors list six advantages:

1 Interpretable

Every step is an explicit function call, and intermediate values can be inspected.

2 Logical

Uses Python's built-in logical and math operators instead of learning them.

3 Flexible

Any vision or language module can be added to the API.

4 Compositional

The code LLM splits the task into small subtasks, solved step by step.

5 Adaptable

Improve any module, and ViperGPT improves directly (e.g., a better depth model).

6 Training-free

No retraining or fine-tuning for each new task.

The intervention study (slide 18) backs point 2: break the Python logic, and accuracy drops.

ViperGPT, Section 1 (list of advantages); Figure 7 for the takeaway. The link to Figure 7 is presenter analysis.

Georgia Tech

Why is it competitive without any training?

Prior modular approaches

Learned module
perception
→
Learned module
reasoning
→
...

Modules and program generator are trained on task data: expensive supervision in the form of programs, and poor generalization to new domains.

ViperGPT

Python function
written by Codex
→
find objects
GLIP
→
ask about them
BLIP-2 / X-VLM

Python logic does the decomposition, so each pretrained model only does the job it is best at. Nothing is trained.

Moving the reasoning into code removes the part that needed supervision.

ViperGPT, Sections 1–2 (prior work "require[s] expensive supervision in the form of programs"). The takeaway is presenter analysis.

Limitations and societal implications

1 Bounded by the API

If no function covers it (“When was this photo taken?”), the program cannot do it.

2 Only as good as its modules

It relies on Codex and every vision model; a wrong find() box flows straight into the answer.

3 Still below supervised models

Best zero-shot everywhere, but far from supervised: 72.0 vs 94.0 (OFA) on RefCOCO.

4 Written blind, run once

Codex never sees the image, and nothing is checked or retried.

Societal implications

+ Inspectable steps and no task-specific training. − Inherits every module's biases, runs generated code (needs a sandbox), and a tidy program can make a wrong answer look trustworthy.

Explicit steps make ViperGPT easy to inspect, but nothing inside it checks the steps.

Limitations 1–2 and 4 follow from the method (Section 3); 3 from Table 1. Societal points are presenter analysis; the paper has no impact section.

Discussion questions

How far can this pipeline generalize?

ViperGPT turns image questions into Python programs. Could the same idea cover arbitrary text or multimodal reasoning tasks?

Is the gap to supervised models a hard barrier?

ViperGPT beats every zero-shot baseline but not supervised models. Would better modules in the API close the gap, or is some supervision needed?

Discussion questions are the presenters'.

CoAct-1

Song et al. · ICLR 2026 · USC, Salesforce AI Research, UW

Computer-using Multi-Agent System with Coding Actions

Frontier paper: from code that answers to code that acts

Georgia Tech

Problem: dozens of clicks, or one script

Example from the paper's intro: *find all images in nested folders, resize them, then zip the folder.*

Clicking through the GUI

1 *Open the file manager*

2 *Open the folder*

3 *Open each sub-folder*

4 *Open an image*

5 *Resize it and save*

6 ↻ Repeat for every image

7 *Back to the top folder*

8 *Right-click → Compress*

9 *Name it, confirm*

≈ 9 steps × every image
each one can misclick

Writing code

```
find photos -name "*.jpg" \
  -exec mogrify -resize 800x600 {} +
zip -r photos.zip photos
```

1 script → *one code action for the whole job*

No icons to find, no menus to open. The interpreter does the repetition.

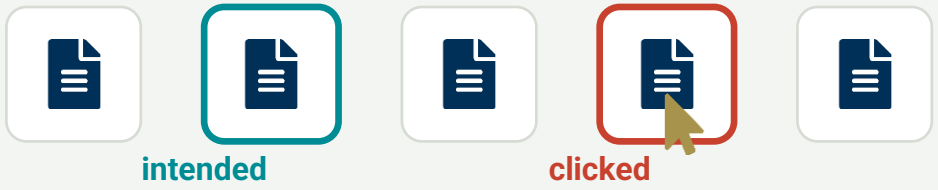
CoAct-1 lets the agent choose: write code when code is easier, click when it isn't.

Georgia Tech

Error propagation: why long chains break

Errors start: look-alike targets

Screens are full of similar icons and menus, so a visual grounding error is easy.



Errors add up

One wrong step can derail every later step. If each action is 95% reliable:

60%	36%	21%
10 actions	*20 actions*	*30 actions*

ViperGPT: *runs its program once. A wrong find() result flows straight into the answer; there is no execution feedback.*

CoAct-1: *checks after every subtask and can retry, but long chains still fail more often (Fig. 3d).*

Fewer, bigger actions lead to fewer chances to fail. That is CoAct-1's bet.

Georgia Tech

Related work: four families of computer-use agents

1

Screen parsing & grounding

Find the right pixel

Locate GUI elements from pixels alone — no DOM or accessibility tree

Examples
OmniParser, SeeClick, Aria-UI, UGround, OS-Atlas

2

Native end-to-end agents

One model does it all

Screenshot in, mouse / keyboard action out — one model

Examples
UI-TARS, OpenCUA, AGUVIS

3

Planner–grounder agents

Split what from where

An LLM planner proposes subgoals; a visual model grounds each click

Examples
GTA-1, Agent S / S2 · infra: AutoGen

4

Hybrid agentic frameworks

Act beyond the GUI

Compose tools and APIs on the fly, at run time

Examples
UFO-2, PyVision, Beyond Browsing, ALITA

CoAct-1 = *planner + executors (3) + code as a tool (4)*

CoAct-1 combines hierarchical planning with code execution to reduce reliance on long GUI sequences.

Georgia Tech

Seminal – Frontier: code becomes an action

$$A = A_{GUI} \cup A_{Code}$$

At every step the agent can either act on the screen or run a script.

GUI action

`click(622, 164)`	*type text*
scroll / drag	*hotkey Ctrl+S*

Best for: *visual apps, browsers, dialogs*

Code action

run a Python script	*run a Bash command*
read / edit files	*change system settings*

Best for: *files, data, settings, batch work*

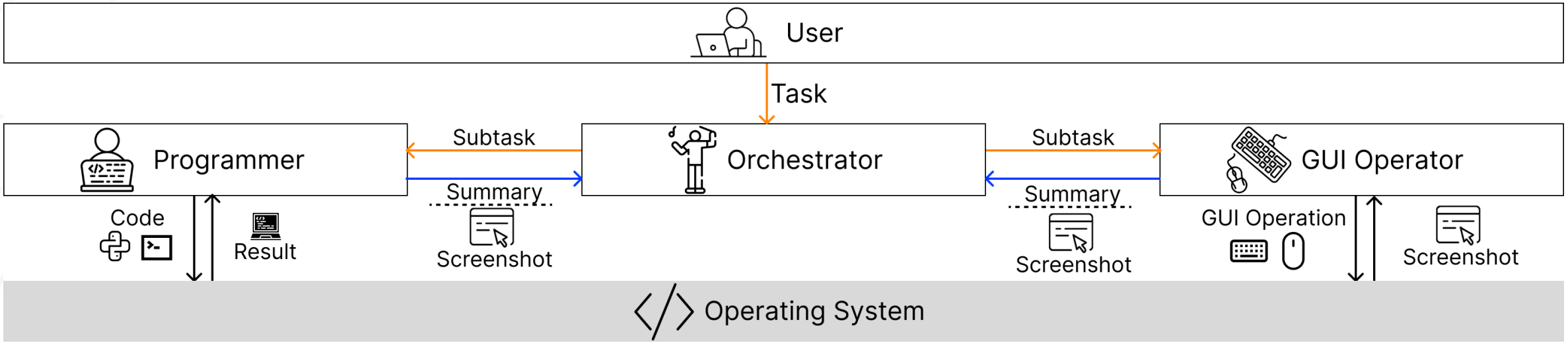
ViperGPT: *code combines visual tools to answer queries*

→

CoAct-1: *code and GUI actions change the computer state*

Georgia Tech

Meet the team: one manager, two workers



Programmer · *the coder*

- *Writes and runs Python or Bash*
- *Sees: terminal output*
- *Model: OpenAI o4-mini*

Orchestrator · *the manager*

- *Splits the goal, picks a worker, checks results*
- *Never touches the computer itself*
- *Model: OpenAI o3*

GUI Operator · *the hands*

- *Clicks, types, scrolls, uses hotkeys*
- *Sees: screenshots*
- *Model: computer-use-preview*

Georgia Tech

How does the manager pick a worker?

No trained router. *Open AI o3 follows rules written in the system prompt:*

1 **File work → code first**

Spreadsheets, files, archives

- For file operations (like modifying spreadsheets), you MUST try the Programmer (call_programmer) first.

2 **Screen work → GUI**

Max 25 actions per call

GUI Operator (call_gui_operator)
- Can interact with the GUI by clicking on a exact position, scrolling, dragging, typing, and using hotkeys.

3 **One worker at a time**

Turns, not parallel

Note: Only call ONE tool (call_programmer or call_gui_operator) per reply.

4 **Manager checks results**

Workers don't grade themselves

- **Do not** let the GUI Operator to do any result check. You need to do it by checking the screenshot yourself.

Georgia Tech

Worked example from the paper

Could you help me unzip the downloaded extension file from /home/user/Desktop/ to /home/user/Desktop/ and configure it in Chrome's extensions?

Programmer

Subtask

Unzip the file /home/user/Desktop/ helloExtension.zip into /home/user/Desktop/

```code
# 1. Go to Desktop
cd /home/user/Desktop || { echo "Desktop directory not found"; exit 1; }
echo "Current directory: $(pwd)"
...
```

Code output:
Current directory: /home/user/Desktop
ZIP archive found: helloExtension.zip
Archive: helloExtension.zip
...
Extraction completed successfully.
Extraction contents: ...

Orchestrator

Round 1 > ***call_coding_agent***

Task:
Unzip the file /home/user/Desktop/ helloExtension.zip into /home/user/Desktop/

Environment:
File path: /home/user/Desktop/helloExtension.zip

LLM Summary

Round 2 > ***call_gui_agent***

Task: Configure the Unpacked Extension in Chrome

Screenshot + GUI Operator Summary

(Omitted Multiple Turns)

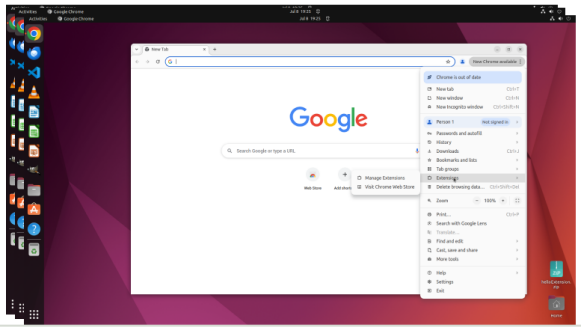
The extension has been successfully added to the chrome and display correctly.
TERMINATE

GUI Operator

Subtask

Configure the Unpacked Extension in Chrome

Navigating to Chrome extensions page
LeftClick(622, 164)
WAIT



1 *Split the request*

2 *Code: unzip the file*

3 *Clicks: add to Chrome*

4 *Check → TERMINATE*

Georgia Tech

Who remembers what?

Manager's notebook

Goal *"Unzip the extension, add it to Chrome"*

Report 1 *Programmer: unzipped to ~/Desktop*

(no screenshot after code — per the prompt)

Report 2 *GUI Operator: extension added*

Screenshot *final screen after the GUI round*

Summarizer
Open AI
o4-mini

short report

Programmer scratchpad

Full chat with the terminal (this subtask only)

Wiped after the subtask

GUI Operator scratchpad

All screenshots + actions (this subtask only)

Wiped after the subtask

The manager reads the report, not the transcript.

Georgia Tech

Checking the work: the old version screen problem

On disk

`report.xlsx`
updated by code ✓

→

On screen

Calc still shows the old version ✗

→

Fix: *the GUI Operator reopens the file, then the manager checks the new screenshot itself.*

The paper's fix, written into the Orchestrator's prompt (Table 5):

- If you used the Programmer to modify a file, have the GUI Operator reopen the file to see the updated results.

- **ALWAYS** check the result through the screenshot by yourself. You can let a GUI Operator to navigate to the correct location for you. After the GUI Operator complete, the screenshot will automatically be returned to you.

Why
GUI self-checks often hallucinate success (App. C.2)

Still a gap: *a screenshot shows only what is in view. An error that scrolled off screen is invisible to the checker (see failure cases).*

Where vision is used, and where it isn't

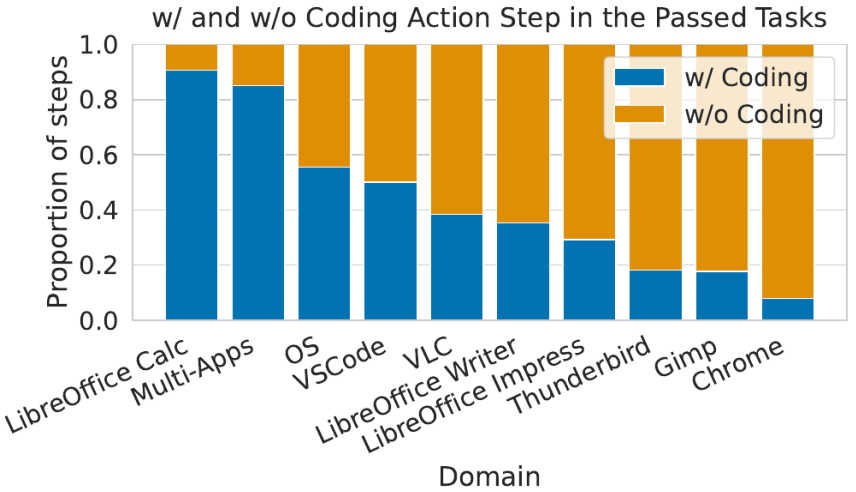
Orchestrator (o3): sees screenshots
plans from them and checks every result itself

GUI Operator (computer-use-preview): sees screenshots
grounds each click or keystroke in pixel coordinates

Programmer (o4-mini): no vision
works from text: subtask, file paths, terminal output

Summarizer (o4-mini): no vision
condenses text logs into a short report

Where CoAct-1 used code in solved tasks (Fig. 3c)



Mostly code: Calc, multi-app. Mostly clicks: Chrome, GIMP, Thunderbird.

Where vision fails: *only what is in view gets checked (reflection error); pixels can lag the file on disk; GUI self-checks hallucinate success.*

Our takeaway: code reduces repeated GUI grounding. Vision remains essential for planning, interface actions and visible-result checks. Limited observations can still hide errors.

Georgia Tech

Experiments: two OS benchmarks

OSWorld

Ubuntu VM · 369 tasks

Office · Daily (Chrome, Thunderbird, VLC) · Professional (GIMP, VS Code) · OS · Multi-app

WindowsAgentArena

Windows VM · 154 tasks

Office · Web · Windows System · VS Code · VLC · Windows Utils

How a task is scored

1 VM starts in a set state
files, apps, websites ready

→

2 Agent gets the goal
the first screenshot

→

3 A script checks the final state
e.g. file exported AND MD5 matches

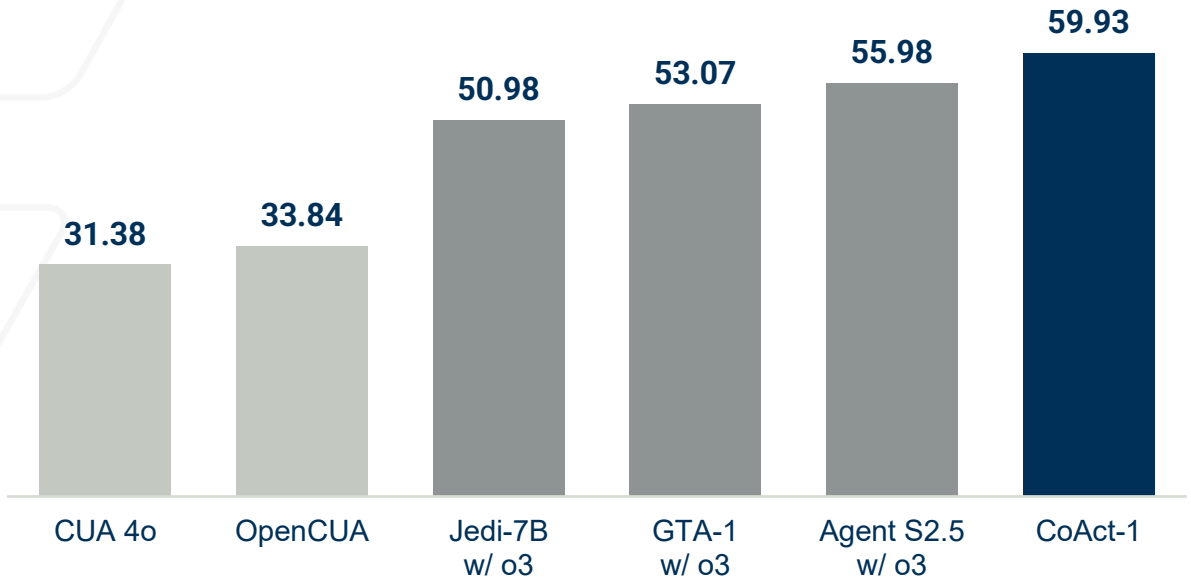
Step budget = *max actions allowed (15 / 50 / 100 / 150).* **Saying "done" earns nothing, only the final state counts.**

Baselines: *end-to-end models (o3, UI-TARS, CUA 4o, OpenCUA) and agentic systems with an o3 planner (Jedi, GTA-1, Agent S2.5).*

Georgia Tech

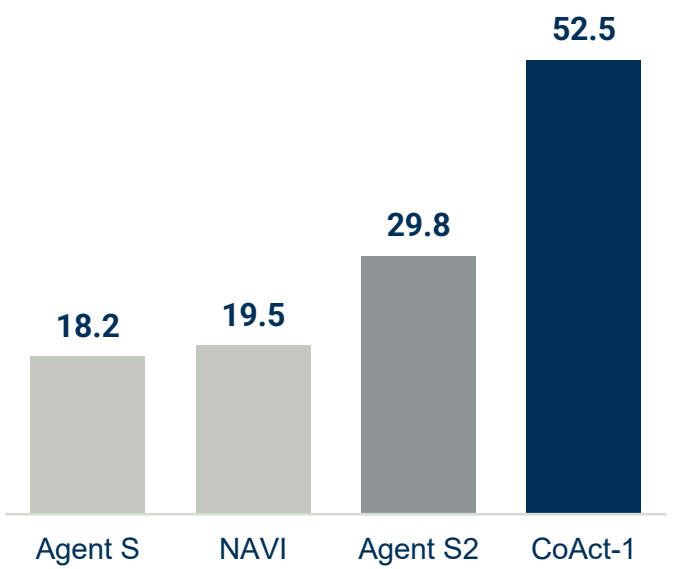
Main results: best on both operating systems

OSWorld success (%), all at 100 steps



60.76% (the abstract's number) is at 150 steps, where no baseline is reported. At 15 steps, Jedi w/ o3 leads (42.37 vs 39.81).

WindowsAgentArena success (%)



83.3%

Windows System

77.7%

Windows Utils

30.4%

Office (weakest)

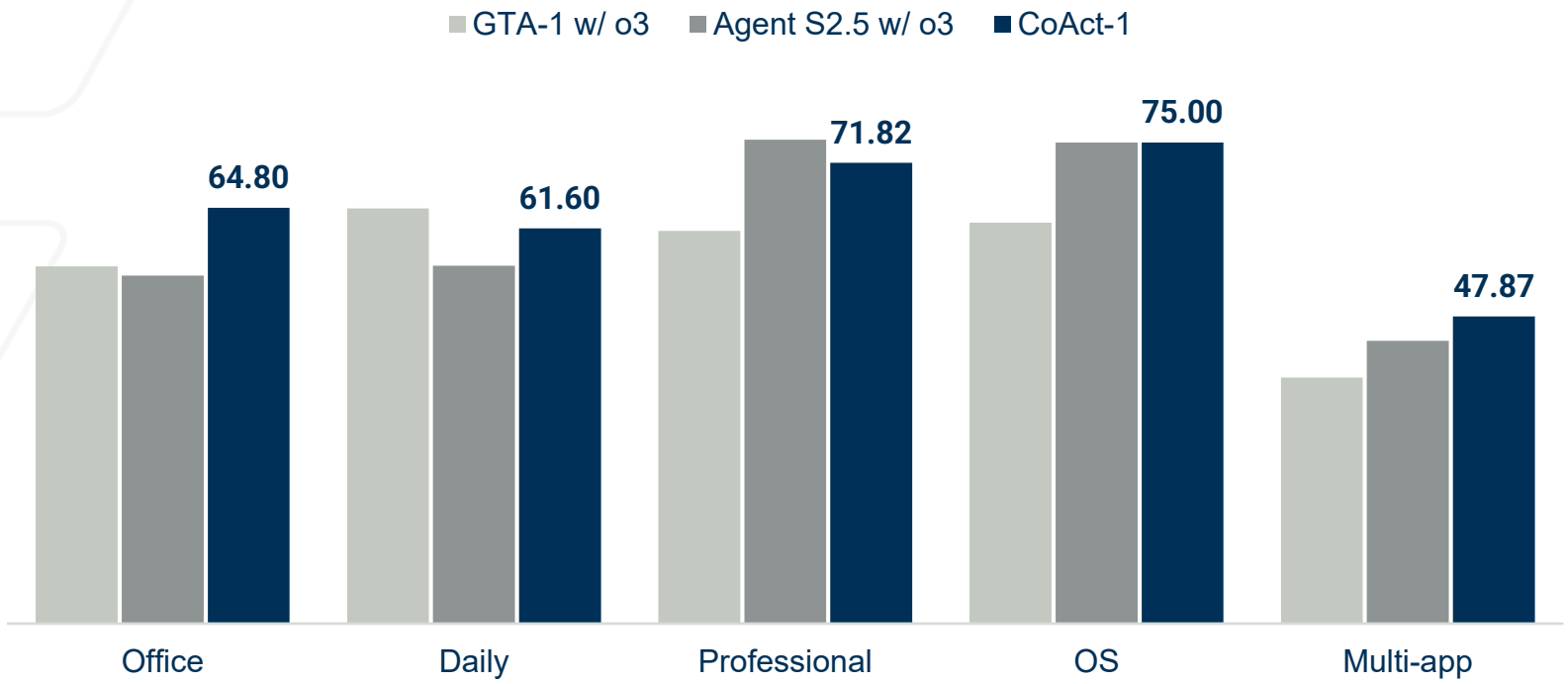
CoAct-1 at 100 steps; the baselines' step budgets are not stated (Table 2).

Our takeaway: CoAct-1 leads Agent S2.5 by 3.95 points at 100 steps. This supports the overall design, but other system differences prevent attributing the gap solely to code.

Georgia Tech

Where does code help most?

Success rate (%) by task category, 100-step budget



Office: +10.6 pts

vs Agent S2.5: file and data work

Multi-app: +3.8 pts

moving data between apps

OS: tie at 75.00

same as Agent S2.5

Visual apps: not best

Daily (Chrome) and Professional (GIMP)

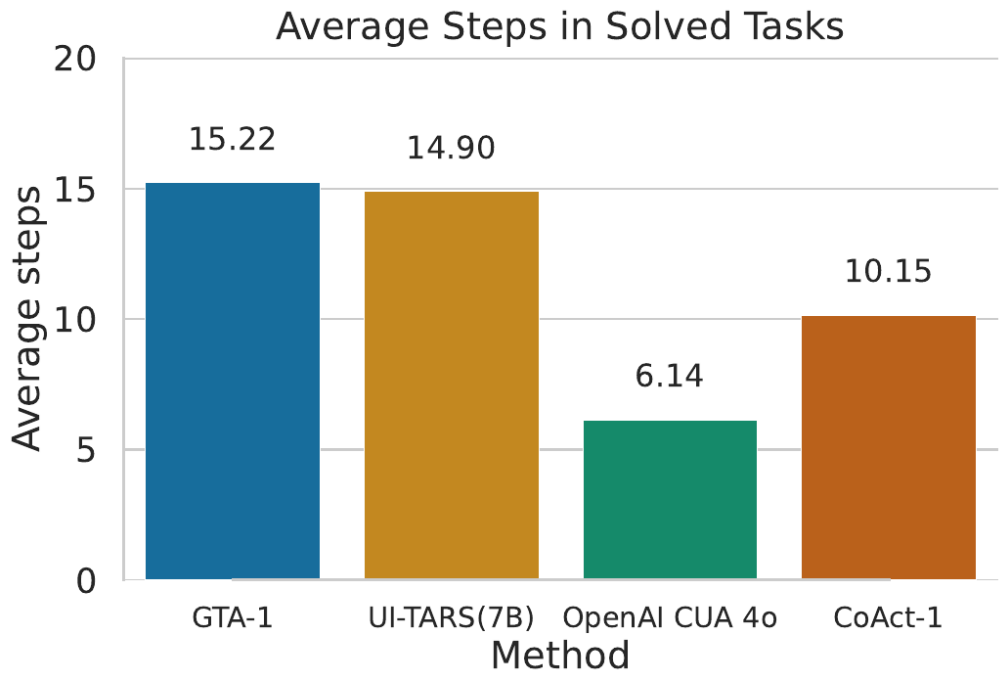
Our takeaway: the category pattern fits the case for code on file and data work. Grounding quality may explain some gaps, but these comparisons do not isolate its contribution.

CoAct-1, Table 1, 100-step block. Labels shown for CoAct-1 only. Takeaway is presenter analysis.

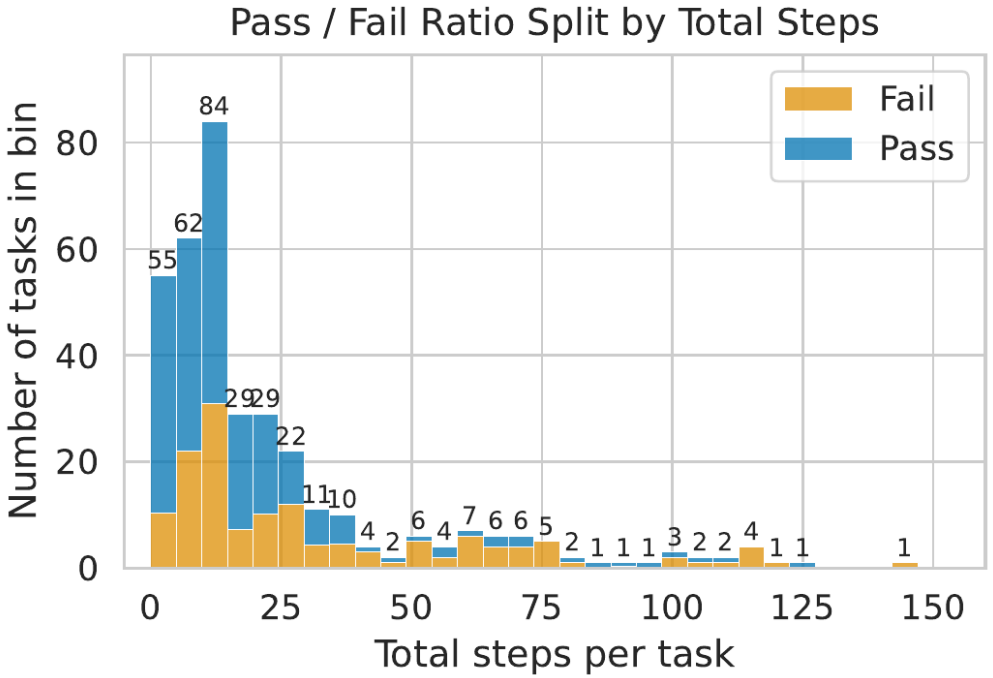
Georgia Tech

Fewer steps, fewer chances to fail

1 CoAct-1 solves tasks in fewer steps



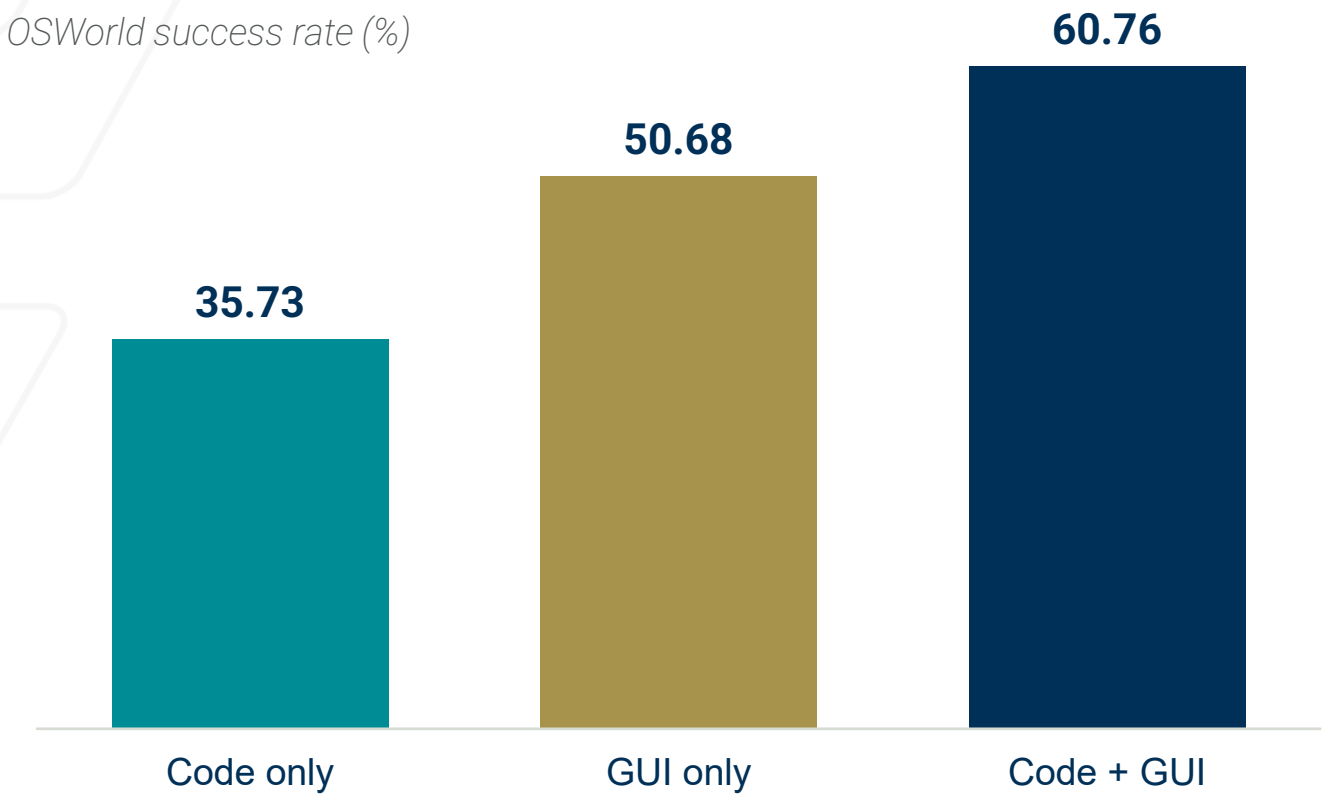
2 Longer tasks fail more often



Our takeaway: one script can replace many GUI actions, but steps differ in work and each method solves a different task set. Time, tokens and dollars are unreported. Fig. 3d shows correlation.

Georgia Tech

Ablation: both tools beat either one alone



Code only 35.73% · 1.14 steps

Fast, but can't do screen-only tasks

GUI only 50.68% · 11.20 steps

Handles more, but slowly

Code + GUI 60.76% · 10.15 steps

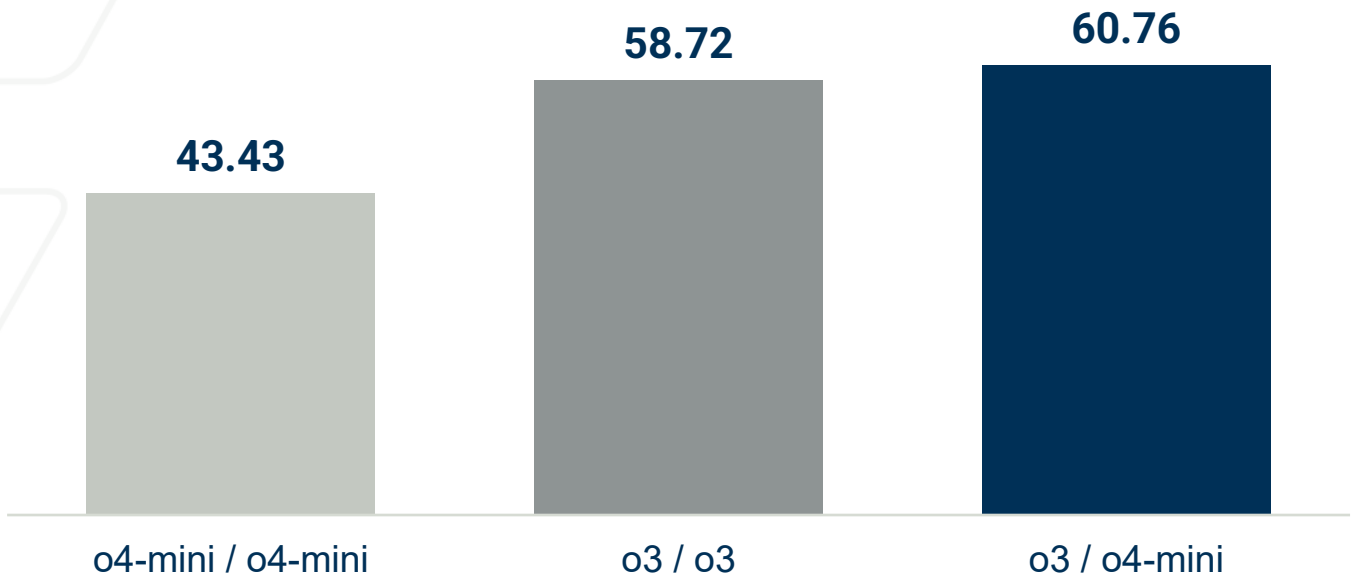
+10.1 pts over GUI only

Our takeaway: hybrid actions improve overall success, but do not lead every category (OS: GUI 79.16, hybrid 75.00). The ablation does not isolate routing quality or prove separate agents are necessary.

Georgia Tech

Orchestrator quality affects performance

OSWorld success (%) — Orchestrator / Programmer



+17.3 pts

Upgrade only the manager (o4-mini → o3), same coder

−2.0 pts

o3 as the coder instead of o4-mini (58.72 vs 60.76)

Our takeaway: among these configurations, upgrading the Orchestrator gives the largest gain. The experiment does not separate visual understanding, planning and instruction following.

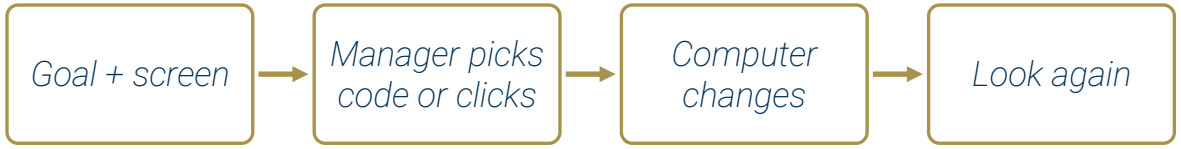
Georgia Tech

What persisted and what changed?

ViperGPT (2023): one pass



CoAct-1 (2026): a loop



Persisted

- *An LLM writes programs that combine tools*
- *Reuses pretrained specialist models, no new training*
- *Explicit intermediate steps you can inspect*

Changed

- *Acts on a changing environment, with side effects*
- *Repeated delegation, memory and summaries*
- *Code is mixed with GUI actions*
- *Evaluates the final computer state*
- *Uses execution feedback for retries and replanning*

Why the shift: *users want finished workflows, and strong coding and GUI-action models now exist.*

Georgia Tech

Strengths and open questions

Strengths

- *Gains on two OSes: OSWorld 60.76%, WindowsAgentArena 52.5%*
- *Ablation isolates the idea: code + GUI beats code-only and GUI-only*
- *Scored by the final computer state, not self-report*
- *Fewer steps per solved task: 10.15 vs 15.22 (GTA-1)*
- *No new training: built from off-the-shelf models*

Still open

- *Would one agent with both tools do as well?*
- *Time, tokens, and cost are not reported*
- *Which memory or checking rules matter? (no ablation)*
- *Depends on specific OpenAI models and prompts*

Supports hybrid actions more directly than the need for multiple agents.

Limitations: When does CoAct-1 fail?

1 Missing concept

"Keep focus on the debug console" → searched "debug", "console"; never linked it to breakpoints, missed the right setting.

2 Ambiguous scope

"Hide __pycache__ folders" → changed the workspace setting instead of the global user setting.

3 Checker can't see it

GUI Operator mistypes a cell, then scrolls away → the final screenshot hides the error.

4 Hallucination

Manager plans around a webpage it never opened; GUI Operator claims it finished when it didn't.

A checker can only catch what it can see.

Georgia Tech

Societal implications

Benefits

- *Less repetitive file and office work*
- *One assistant that works across many apps*
- *Fewer steps per task, fewer chances to slip*

Risks

- *One script can change many files at once*
- *Screenshots can expose private data*
- *Agents are given the system (sudo) password in their prompts*
- *Results come from sandboxed VMs, not personal PCs*

The paper calls for permission controls and privacy protection before real deployment.

Connections to course themes

Grounding

ViperGPT: *grounding is a module: find() calls GLIP.*
CoAct-1: *code avoids some GUI grounding; screenshots still support planning, actions and checks.*

Chain of thought

ViperGPT: *the program is a "programmatic chain-of-thought" (its Fig. 5).*

CoAct-1: *the manager's prompt fixes the steps: describe, plan, execute, verify.*

Reinforcement learning

ViperGPT: *no extra task-specific policy training.*
CoAct-1: *feedback changes context and plans at inference time. The framework does not update model weights.*

Hallucination & robustness

ViperGPT: *module errors can propagate despite valid code and local error handling.*
CoAct-1: *verification can miss hidden errors or unsupported completion claims.*

Open question (ours): *could RL learn the routing policy π_orch instead of a hand-written prompt?*

Takeaways

1

Use the right tool for each step

Code for files and data, clicks for visual apps.

2

Mixing both beats either alone

60.76% vs 50.68% (GUI only) on OSWorld.

3

Knowing a task is really done is still hard

A check is only as good as what the checker can see.

Discussion questions

Would one strong agent with both GUI and code tools match this three-role team?

What would a fair comparison hold fixed (models, prompts, step budget, cost?)

What evidence should an agent collect before it says a task is done?

Screenshot? File difference? Cell-by-cell check? How much checking is worth the cost?

References

Song, L., Dai, Y., Prabhu, V., Zhang, J., Shi, T., Li, L., Li, J., Savarese, S., Chen, Z., Zhao, J., Xu, R., Xiong, C. CoAct-1: Computer-using Multi-agent System with Coding Actions. ICLR 2026. arXiv:2508.03923v3.

Surís, D., Menon, S., Vondrick, C. ViperGPT: Visual Inference via Python Execution for Reasoning. ICCV 2023. arXiv:2303.08128.

Andreas, J., Rohrbach, M., Darrell, T., Klein, D. Neural Module Networks. CVPR 2016.

Gupta, T., Kembhavi, A. Visual Programming: Compositional Visual Reasoning without Training. CVPR 2023. arXiv:2211.11559.

Xie, T., et al. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. 2024. arXiv:2404.07972.

Bonatti, R., et al. Windows Agent Arena: Evaluating Multi-Modal OS Agents at Scale. 2024. arXiv:2409.08264.

Minderer, M., et al. Simple Open-Vocabulary Object Detection with Vision Transformers (OWL-ViT). ECCV 2022. arXiv:2205.06230.

Yao, Y., et al. Qwen3-VL-Seg: Unlocking Open-World Referring Segmentation with Vision-Language Grounding. 2026. arXiv:2605.07141.

Zhang, Z., et al. Multimodal Chain-of-Thought Reasoning in Language Models. TMLR 2024. arXiv:2302.00923.

Aneja, J., et al. Phi-4-reasoning-vision-15B Technical Report. 2026. arXiv:2603.03975.

DeepSeek-AI. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. 2025. arXiv:2501.12948.

Wang, Z., et al. Perception-Aware Policy Optimization for Multimodal Reasoning (PAPO). ICLR 2026. arXiv:2507.06448.

Guan, T., et al. HallusionBench: Entangled Language Hallucination and Visual Illusion in Large Vision-Language Models. CVPR 2024. arXiv:2310.14566.

Shukla, T., Kira, Z. SAGE: Sink-Aware Grounded Decoding for Multimodal Hallucination Mitigation. 2026. arXiv:2603.27898.

Code: github.com/SalesforceAIResearch/CoAct-1

Georgia Tech