TALONS
LABORATORIES

# UI-TARS Desktop Agent POC Documentation

**Version:** 2.0
**Last Updated:** October 21, 2025

# Best Practices

**Critical: Cursor Visibility and UI Focus**

- **Keep the cursor visible** in the window during automation. The model uses the cursor position to determine coordinates more accurately.
- Ensure the **target application window has focus** when screenshots are taken.
- **Avoid moving the mouse** during automation steps to prevent confusing the model.
- If the agent seems to misidentify UI elements, try **moving the cursor near** (but not directly on) the element you want to interact with.
- Use the `**minimize_all_windows()**` function to clear any existing window to begin a fresh desktop automation (For using CLI script only, call this function in run_with_arguments.py or run_agent_loop.py)

# Architecture Overview

How the current pipeline maps to UI-TARS desktop logic:

| **UI-TARS (TS code)** | **Python script** |
|---|---|
| `prompts.ts` (system + action space + user instr) | `prompts.py` (`get_simple_system_prompt`, `get_detailed_user_prompt`) |
| `action-parser/*` | `action_parser.py` |
| `ComputerOperator.ts` | `desktop_controller.py` |
| `runAgent.ts` (loop + retry + phase mgmt + history) | `desktop_agent_code.py` |
| App entry (`index.tsx`, renderer, UI) | `run_with_arguments.py` (CLI wrapper for CL arguments) `run_desktop_loop.py` (CLI wrapper for instructions in a text file) |

The goal is to replicate the logic almost 1:1, just without the GUI.

## Core Components

### `action_parser.py`
This module parses the model's output into structured action objects.

### `desktop_controller.py`
Handles all desktop interactions using PyAutoGUI.

### `desktop_agent_core.py`
The brain of the operation that manages the agent's state and loop.

### `prompts.py`
Defines the system and user prompts for the UI-TARS model.

### `run_with_arguments.py`
CLI wrapper for running single instructions.

```
Usage: python run_with_arguments.py "Your instruction here"

Optional: --session-id to specify a session ID
```

### `run_agent_loop.py`
Processes and runs multiple instructions from a text file:

```
Usage: python run_agent_loop.py instructions.txt

The argument is the path to the text file

Optional: --otp for OTP code and --mobile for mobile number
```

# Usage Guide

## Command Line Usage

The POC provides two main command-line interfaces:

| Mode | Session Behavior | How Instructions Run | Usage Example |
|---|---|---|---|
| Single instruction | Auto timestamp folder (or shared if `--session-id`) | Atomic, stops when finished | `python run_with_arguments.py "Open chrome"` |
| Instruction file | Unique session ID for each instruction | Extract and run each instruction separately from a text file | `python run_agent_loop.py instructions.txt` |

**Note that the instruction text file must be separated by a new line to be recognised as individual instructions.*

## Exit Codes

The scripts use consistent exit codes to communicate status for integration with PA flow:

| Exit Code | Meaning | Action |
|---|---|---|
| 0 | Success | Instruction completed successfully |
| 1 | User intervention | Manual action required |
| 2 | Agent error | Error in execution or parsing |
| 3 | Authentication required | Requires phone number or OTP from user |
| 130 | User cancelled | Operation interrupted by user (Ctrl+C) |

## Single instruction - run_with_arguments.py

**Purpose:** Executes a single instruction with logging and session management.

**Usage:**

python run_with_arguments.py "instruction text"

python run_with_arguments.py "instruction text" --session-id <session_id>

**Features:**

- Creates session directory structure:
  - Without `--session-id`: `session/session_TIMESTAMP/`
  - With `--session-id`: `session/session_ID/session_TIMESTAMP/`
- Logs all output to `session_log.txt`
- Returns appropriate exit codes based on agent status

**Session Directory Contents:**

- `session_log.txt`: Complete execution log
- `screenshot_TIMESTAMP.png`: All captured screenshots

## Batch Orchestrator - run_agent_loop.py

**Purpose:** Executes multiple instructions sequentially from a file.

**Usage:**

python run_agent_loop.py <instruction_file>

python run_agent_loop.py <instruction_file> --otp <otp_value> --mobile <mobile_number>

**Features:**

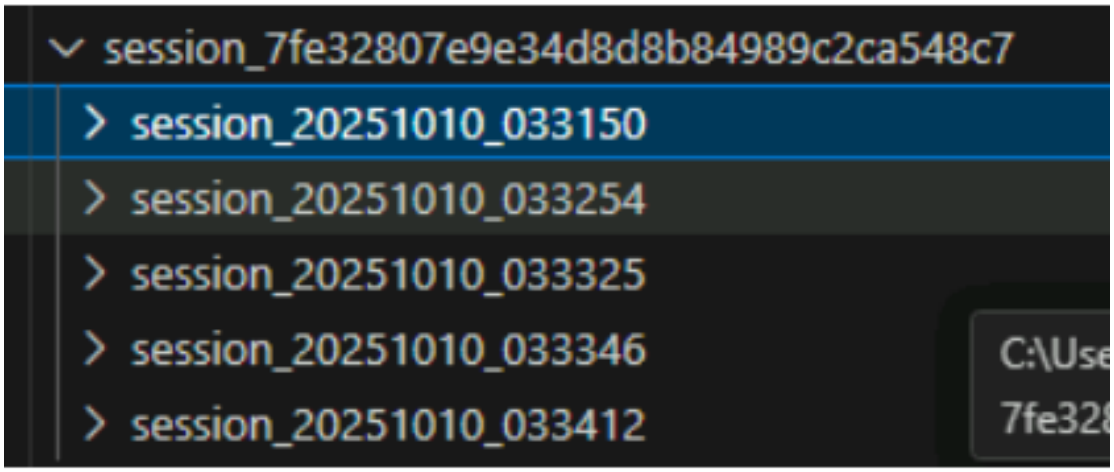
- Reads instruction file and splits by empty lines
- Creates a parent session ID for all instructions
- Supports dynamic variable replacement:
  - `$Number` → replaced with `--otp` value
  - `$Mobile` → replaced with `--mobile` value
- Handles exit codes and stops on errors

## Session Management

Sessions are organised as follows:

```
└── session_<ID>/                    # Parent folder with unique ID
    ├── session_<TIMESTAMP>/         # Child folder per individual instruction
    │   ├── session_log.txt
    │   ├── screenshot_<TIMESTAMP1>.png
    │   ├── screenshot_<TIMESTAMP2>.png
    │   └── ...
    └── session_<TIMESTAMP2>/
```

Sample folder structure:



## Sample Session logs

```
--- [Step] Calling UI-TARS Model —

Server response received.

Thought: I noticed the icon for Forensic Email Collector on the
desktop, and it's time to launch it. This is the first step in
completing the task, so I need to double-click on this icon to open
the program. The icon is located on the left side of the desktop, and
I can see it clearly.

Action: {'action': 'left_double', 'params': {'start_box': [36, 347]}}


--- [Step] Calling UI-TARS Model —

Server response received.

Thought: The program interface is now fully displayed in front of me,
and I can see the input box for the email address. The next step is to
enter the email address automationtest622@gmail.com into this input
box. I need to click on the input box to activate it so that I can
start typing.

Action: {'action': 'click', 'params': {'start_box': [707, 475]}}
```

# Prompt Optimisation Module

The Prompt Optimisation Module is a standalone Python utility that transforms user's lengthy and unclear instructions into structured, atomic steps optimised for UI automation. It currently calls the UI-TARS 1.5-7B model to refine instructions, making them more suitable for automated execution. (Note that the model can be swapped to any model of your choice by changing the `API_URL` and `model` name in the `prompt_optimiser.py` file)

**Important Note:** When using command-line input with special characters (e.g., passwords with `$`, `@`), use the `--file` argument or escape special characters to avoid shell interpretation issues.

## Usage

The module can be used in two ways:

1. **Direct Text Input**:

   ```
   python prompt_optimiser.py "Open Forensic Email Collector and enter email address automationtest622@gmail.com"
   ```

2. **File Input**:

   ```
   python prompt_optimiser.py --file instructions.txt
   ```

3. **Custom Output Path** (optional):

   ```
   python prompt_optimiser.py "Your instructions" --output custom_output.txt
   ```

Example:
`python prompt_optimiser.py "Open chrome and search for ui-tars github, go into the repo and click the issues tab, search for open issues related to action space"`

# Power Automate Desktop Integration

The flow is crafted to integrate the python script with proper entry points and minimal user intervention. The instruction file is split into half, one before 2FA and one after.

## Known Limitations and Constraints

**Model Performance Degradation via Power Automate (PA)**

- **Issue**: Model performance is noticeably reduced when scripts are executed through Power Automate compared to direct execution
- **Impact**: Potentially different behavior patterns
- **Root Cause**: Unknown - may be related to:
    - Network latency and connection overhead
    - Resource allocation differences in PA environment
    - Different execution context or environment variables

**Safe Exit Limitations in PA Environment**

- **Issue**: Ctrl+C interrupt handling does not work when scripts are executed via Power Automate
- **Impact**: Cannot gracefully terminate long-running scripts through standard interrupt signals

**Mitigation**: Consider direct script execution for optimal model performance

## Basic Flow Implementation

## 2 Subflows for each set of instructions

Main flow:

```
# Initialize variables
```

Set Variable: FirstFile = instructions.txt (First half of the instruction)

# Main instruction loop

Minimize PA Window to clear screen

Minimize Window: "Power Automate"

# Run Python script

Run DOS Command:

Command: C:\path\to\python.exe run_with_arguments.py %FirstFile%
Store exit code in: %ExitCode%

# Handle exit codes

If %ExitCode% = 0 (Success)

Display Message: "Instructions are completed."

Else If %ExitCode% = 1 (User Intervention)

Display Message: "Agent paused. Please perform manual steps."

Else (Any other exit code)

Display Message: "Error occurred (code %ExitCode%). Operation will end."

(Phase_2 flow is exactly the same except the instruction path is set to SecondInstruction)