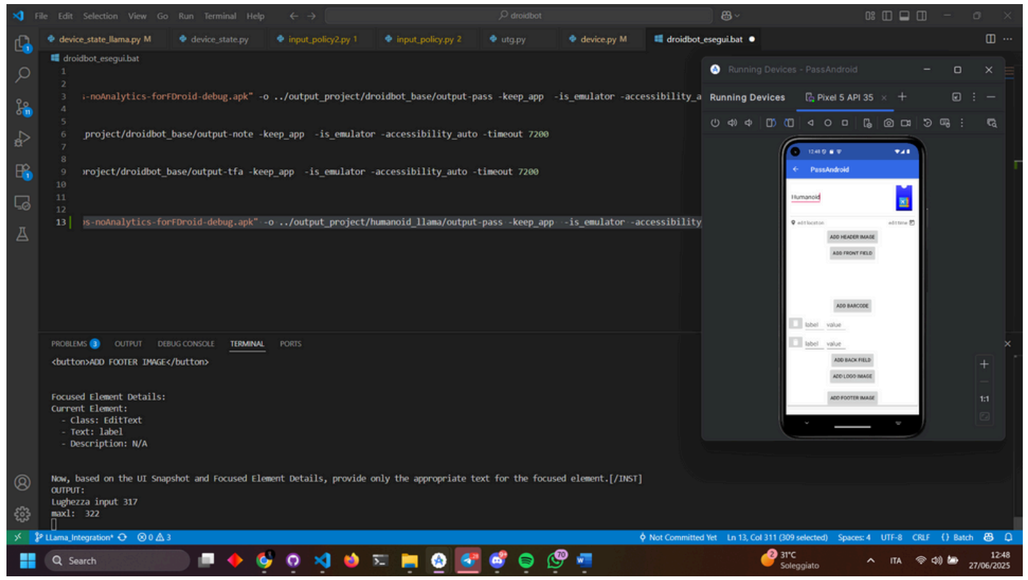
POLITECNICO DI TORINO

Large Language Models 2024/2025

Context-Aware GUI Testing con LLMs



Presentation by : A3

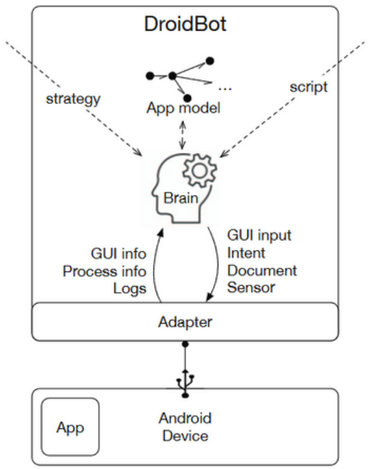
Michele Russo, Raffaele Martone, Giuseppe Monteasi

Main GUI Test Framework

02

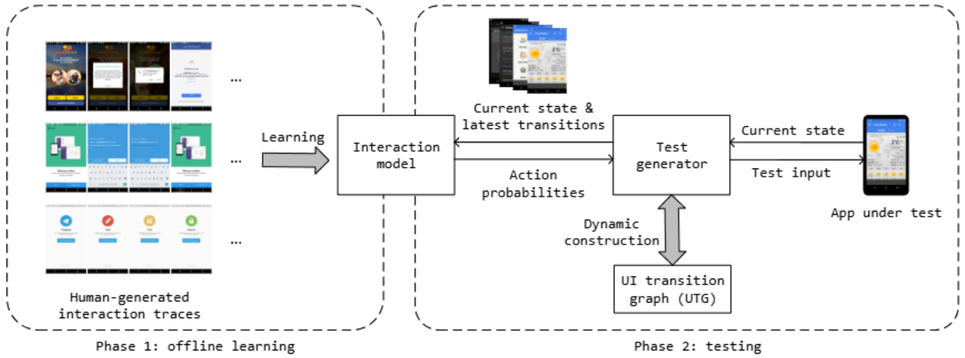
DROIDBOT

DroidBot is a lightweight, open-source tool for automated test input generation on Android apps. Unlike random event generators, it analyzes the UI hierarchy and device state in real time, building a dynamic graph of UI states and transitions. This enables systematic, deep exploration and richer coverage, making DroidBot effective at uncovering behaviors that only appear under specific conditions.



HUMANOID

Humanoid is an advanced input generator that integrates deep learning to simulate realistic user interactions. Trained on large-scale human interaction data, it predicts which UI elements real users would select and what actions they would take. By combining visual feature extraction and sequence modeling, Humanoid explores the app more like a real user, reaching GUI states often missed by traditional or random approaches.

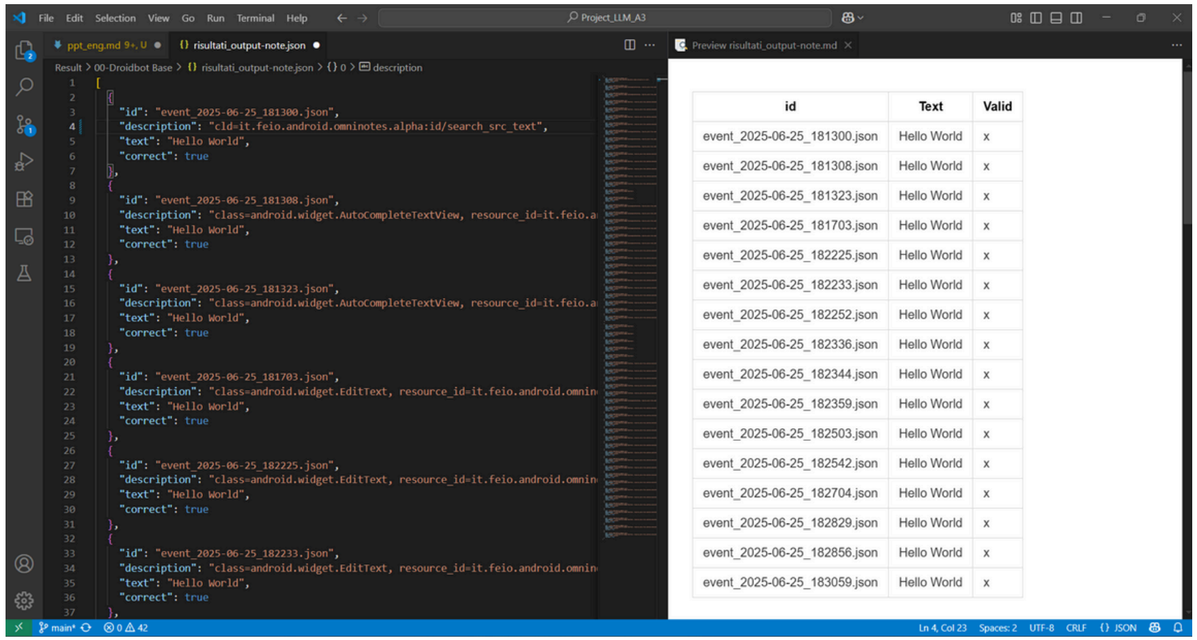


The Core Problem

Traditional mobile testing frameworks have a critical limitation

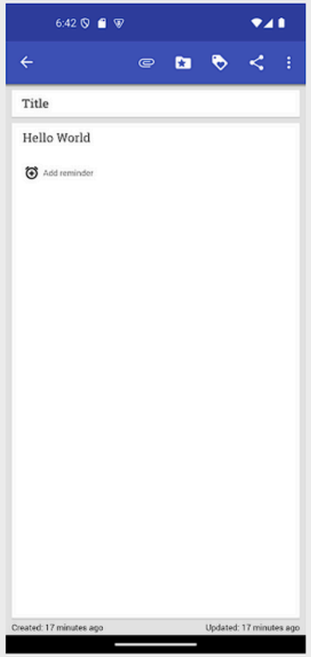
Main problem

- **DroidBot** and **Humanoid** only insert "Hello World" in text fields or pre-setted input
- Inability to overcome screens requiring specific inputs
- Limited coverage of real app functionalities



id	Text	Valid
event_2025-06-25_181300.json	Hello World	x
event_2025-06-25_181308.json	Hello World	x
event_2025-06-25_181323.json	Hello World	x
event_2025-06-25_181703.json	Hello World	x
event_2025-06-25_182225.json	Hello World	x
event_2025-06-25_182233.json	Hello World	x
event_2025-06-25_182252.json	Hello World	x
event_2025-06-25_182336.json	Hello World	x
event_2025-06-25_182344.json	Hello World	x
event_2025-06-25_182359.json	Hello World	x
event_2025-06-25_182503.json	Hello World	x
event_2025-06-25_182542.json	Hello World	x
event_2025-06-25_182704.json	Hello World	x
event_2025-06-25_182829.json	Hello World	x
event_2025-06-25_182856.json	Hello World	x
event_2025-06-25_183059.json	Hello World	x

State Details



package	it.feio.android.omninotes.alpha
activity	it.feio.android.omninotes.MainActivity
state_str	0bc2e9c92f98570689d2f29265de1f99
structure_str	f835bff3c709da412469d04183af13e3

Research Questions

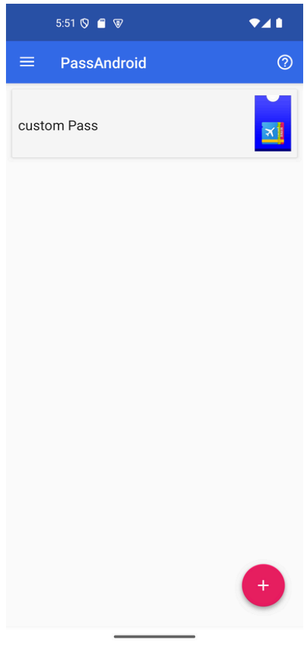
Focus
Measuring effectiveness (working tests) vs coverage (explored states)

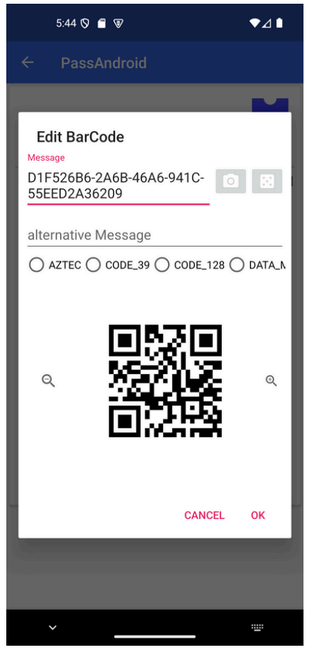
- **RQ1**: How effectively can LLMs generate working context-aware test cases for mobile GUI testing?
- **RQ2**: What coverage can be obtained by generating test cases with LLMs?

3 Tested Apps

PASSANDROID

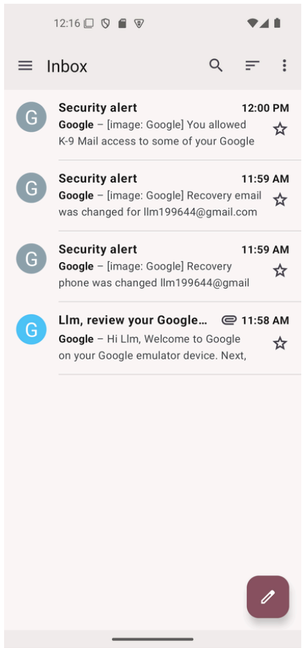
Digital pass management





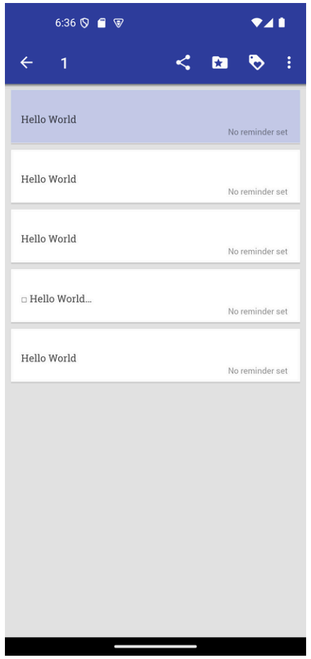
THUNDERBIRD

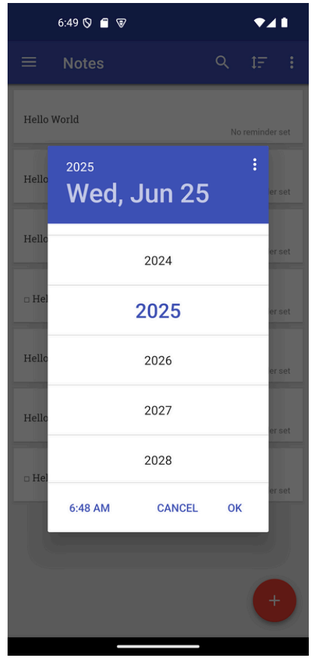
Email



OMNI-NOTES

Note-taking

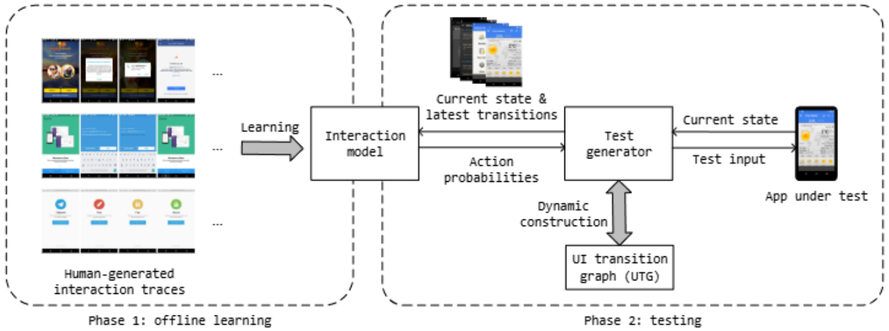




Methodology - The 7 Configurations

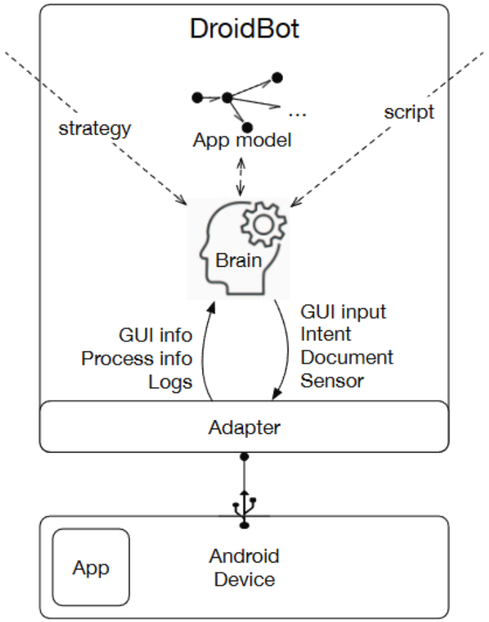
Baseline

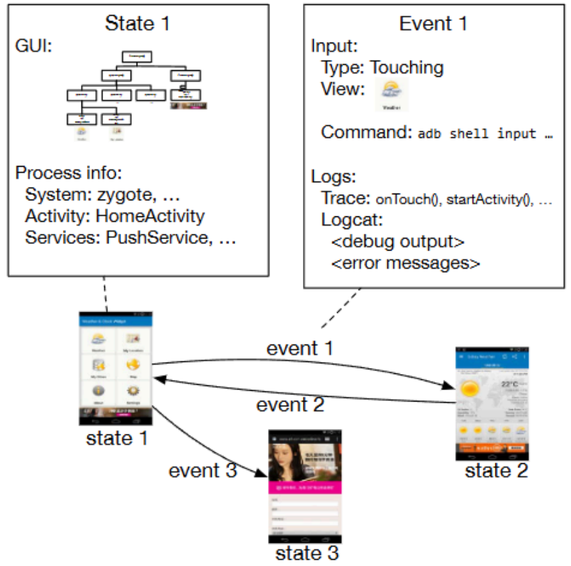
- Droidbot Base
- Humanoid Base



Replay-Enhanced

- Droidbot + Replay output (offline approach)
- Humanoid + Replay output (offline approach)





LLM-Enhanced (Llama-3.2-1B-Instruct)

- Droidbot + LLAMA for textual input
- Droidbot + LLAMA for textual input + Choice
- Humanoid + LLAMA for textual input

LLM Integration - Two Approaches

Online Approach (Real-time):

- Direct integration into DroidBot/Humanoid code
- Dynamic generation during exploration
- Prompt engineering with few-shot learning

Offline Approach (Replay):

- Post-processing DroidBot JSON files
- Replacing "Hello World" with contextual texts
- Re-running modified tests

```
Prompt sent to Llama:
<s>[INST] <<SYS>>You are an assistant that generates only a realistic, context-appropriate text input for a UI element. Respond with a single action to take, no explanations, no formatting, no extra sy
mbols.<</SYS>>

### Few-Shot Examples:

Example 1:
[UI Representation]
<input id=0 text='Email address' bound_box=10,50,310,100>Enter email</input>
Possible Actions:
0: event_type=touch, desc='Show Calendar List and Settings drawer', resource_id=None, class=android.widget.ImageButton, bounds=[[11, 147], [165, 301]]1: event_type=touch, desc='July', resource_id=com.g
oogle.android.calendar:id/date_picker_button, class=android.widget.LinearLayout, bounds=[[166, 147], [650, 301]]2: event_type=touch, desc='Search', resource_id=com.google.android.calendar:id/action_sea
rch, class=android.widget.Button, bounds=[[661, 158], [793, 290]]3: event_type=touch, desc='Jump to Today', resource_id=com.google.android.calendar:id/action_today, class=android.widget.Button, bounds=
[[793, 158], [925, 290]]Output:
1UI Snapshot:
<button>July</button>


Focused Element Details:
Current Element:
  - Class: FrameLayout
  - Text: N/A
  - Description: N/A


Possible Actions:
0: event_type=touch, desc='Show Calendar List and Settings drawer', resource_id=None, class=android.widget.ImageButton, bounds=[[11, 147], [165, 301]]
1: event_type=touch, desc='July', resource_id=com.google.android.calendar:id/date_picker_button, class=android.widget.LinearLayout, bounds=[[166, 147], [650, 301]]
2: event_type=touch, desc='Search', resource_id=com.google.android.calendar:id/action_search, class=android.widget.Button, bounds=[[661, 158], [793, 290]]
3: event_type=touch, desc='Jump to Today', resource_id=com.google.android.calendar:id/action_today, class=android.widget.Button, bounds=[[793, 158], [925, 290]]
4: event_type=touch, desc='Signed in as llm llm199644@gmail.com
```

Prompt Engineering

08

Text Input Prompt

This prompt instructs the LLM to generate a realistic, context-aware text input for a UI element, based on the interface structure and properties. It uses few-shot examples and requires the output to be concise, context-matching, and without any extra formatting or symbols.

```python
def ask_llama(self,view, maxl=200, temp=0.7, full_view_representation=None):
    rappresentation = ""
    for line in full_view_representation.split('\n'):
        type = line.split('<')[1].split(' ')[0]
        line = line.split('>')[1].split('<')[0]
        if len(line) > 0:
            rappresentation += f"<{type}>" + line + "</" + type + ">\n"
    full_view_representation = rappresentation
    # Generate a brief description of the current element based on available attributes
    view_text = self.__safe_dict_get(view, 'text', default='').strip()
    content_description = self.__safe_dict_get(view, 'content_description', default='').strip()
    view_class = self.__safe_dict_get(view, 'class', default='').split('.')[-1]

    # Build a readable summary for the current view
    view_details = (
        f"Current Element:\n"
        f"  - Class: {view_class}\n"
        f"  - Text: {view_text if view_text else 'N/A'}\n"
        f"  - Description: {content_description if content_description else 'N/A'}\n"
    )

    # Construct a structured prompt that incorporates both the global UI representation
    # and the details of the active view
    prompt = (
            f"<s>[INST] <<SYS>>"
            "You are an assistant that generates only a realistic, context-appropriate text input for a UI element. "
            "Respond with a single word of plain text, no explanations, no formatting, no extra symbols."
            "<</SYS>>\n\n"
            "### Few-Shot Examples:\n\n"
            "Example 1:\n"
            "[UI Representation]\n"
            "<input id=0 text='Email address' bound_box=10,50,310,100>Enter email</input>\n"
            "Output:\n"
            "john.doe@example.com\n\n"
```

```
<s>[INST] <<SYS>>
You are a helpful assistant designed to generate context-aware
     text inputs for UI elements.
Analyze the interface structure, element properties, and
    existing content to suggest realistic values.
<</SYS>>
### Task Description:
1. Given the current UI state and a focus element (marked as
    editable):
2. Suggest a text input that matches the element's context.
3. The output MUST be few words, without any additional
    explanations or formatting or symbols.
4. The output MUST be a single line of text.
5. The output MUST be different from previous texts.
5. Follow these guidelines:
- Use actual data formats (emails, names, numbers) when
    detectable
- Mirror the style of existing content when applicable
- Prioritize content descriptions over placeholder texts
- Keep inputs minimal but meaningful

### Few-Shot Examples:
Example 1:
[UI Representation]
<input id=0 text='Email address' bound_box=10,50,310,100>Enter
     email</input>
Output:
john.doe@example.com

Example 2:
[UI Representation]
<input id=1 text='Search bar' bound_box=20,100,300,150>Search
    products...</input>
<button id=2 text='Search' bound_box=320,100,620,150></button>
Output:
wireless headphones

### Current Interface:
{full_view_representation}
### Focus Element Details:
{view_details}

### Instruction:
Based on the above context, suggest appropriate text for the
    focus element.
Provide only the raw text input without explanations, symbols
    or formatting, JUST TEXT. [/INST]

OUTPUT:
```

Prompt Engineering

Action Selection Prompt:

This prompt asks the LLM to select the most appropriate action for the current UI state from a list of possible actions, using context from the UI snapshot, focused element details, and action history. The output must be only the action index, with no explanations or formatting

```python
def ask_choice_llama(self,view, maxl=200, temp=0.7, full_view_representation=None,possible_actions=None):
    rappresentation = ""
    for line in full_view_representation.split('\n'):
        if '<' in line and '>' in line:
            try:
                type = line.split('<')[1].split(' ')[0]
                content = line.split('>')[1].split('<')[0]
                if len(content) > 0:
                    rappresentation += f"<{type}>{content}</{type}>\n"
            except Exception as e:
                print(f"WARNING: errore nel parsing della riga: {line} -- {e}")
        else:
            # se vuoi ignorare le righe senza tag, lascia pure vuoto
            continue

    full_view_representation = rappresentation
    # Generate a brief description of the current element based on available attributes
    view_text = self.__safe_dict_get(view, 'text', default='').strip()
    content_description = self.__safe_dict_get(view, 'content_description', default='').strip()
    view_class = self.__safe_dict_get(view, 'class', default='').split('.')[-1]

    # Build a readable summary for the current view
    view_details = (
        f"Current Element:\n"
        f"  - Class: {view_class}\n"
        f"  - Text: {view_text if view_text else 'N/A'}\n"
        f"  - Description: {content_description if content_description else 'N/A'}\n"
    )
```

```
<s>[INST] <<SYS>>
You are an assistant that generates only a realistic, context-
    appropriate text input for a UI element. Respond with a
    single action to take, no explanations, no formatting, no
    extra symbols.
<</SYS>>

### Few-Shot Examples:

Example 1:
[UI Representation]
<input id=0 text='Email address' bound_box=10,50,310,100>Enter
     email</input>
Possible Actions:
0: event_type=touch, desc='Show Calendar List and Settings
    drawer', resource_id=None, class=android.widget.
    ImageButton, bounds=[[11, 147], [165, 301]]
1: event_type=touch, desc='July', resource_id=com.google.
    android.calendar:id/date_picker_button, class=android.
    widget.LinearLayout, bounds=[[166, 147], [650, 301]]
2: event_type=touch, desc='Search', resource_id=com.google.
    android.calendar:id/action_search, class=android.widget.
    Button, bounds=[[661, 158], [793, 290]]
3: event_type=touch, desc='Jump to Today', resource_id=com.
    google.android.calendar:id/action_today, class=android.
    widget.Button, bounds=[[793, 158], [925, 290]]
Output:
1

UI Snapshot:
{full_view_representation}

Focused Element Details:
{view_details}

Possible Actions:
{view_possible_actions}

History of Actions:
{view_history_actions}

Now, based on the UI Snapshot, Focused Element Details,
    possible actions and History of Actions, provide only the
    appropriate index action.
[/INST]
OUTPUT:
```

PassAndroid Results

Key Findings

- Both DroidBot and Humanoid simulate realistic user interactions: creating digital passes, filling in fields (title, barcode, header), printing, sharing, editing, and accessing the help menu.
- Baseline configurations (DB, H, Replay) explore few states and activities, with no valid text inputs.
- Integrating LLAMA for text generation (DBL, HL) increases state/edge exploration but does not significantly improve valid text input.
- Only when LLAMA is used for both text and action selection (DLC, HL) do we see more valid text inputs and slightly higher activity coverage (up to 5/15).
- Main limitations: low percentage of valid text fields, UI Automator struggles with menu navigation and external file uploads, limiting full activity coverage.

5.1.2 Coverage and Effectiveness

Metric	DB	DBR	DBL	DLC	H	HR	HL
UTG state	110	110	214	77	170	170	119
UTG edge	200	200	2087	232	360	360	230
Activity coverage	4/15	4/15	4/15	5/15	4/15	4/15	5/15
Valid set text	0/66	0/66	0/31	7/182	0/224	0/224	8/289

Table 1: Coverage and effectiveness metrics for PassAndroid across all configurations.

Legenda: **DB**: DroidBot base, **DBR**: DroidBot + Replay output, **DBL**: DroidBot + LLAMA (text generation), **DLC**: DroidBot + LLAMA + Choice (decision making), **H**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLAMA

Thunderbird Android Results

Key Findings – TFA Pre-Login

- No method was able to automatically bypass the login screen; exploration was limited to the app's initial screens.
- Activity coverage and valid text input were both very low across all configurations.
- Integrating LLAMA for text generation did not yield significant improvements due to limited contextual information pre-login.

Key Findings – TFA Post-Login

- After manual login, all methods explored significantly more states and transitions, enabling deeper app interaction.
- Contextual text generation via LLAMA greatly increased the number of valid text inputs.
- The LLAMA + Choice approach further improved text input effectiveness, demonstrating the benefit of LLM-based selection in richer contexts.

5.2.2 Coverage and Effectiveness

Metric	D	DR	DL	DLC	H	HR	HL
UTG state	356	356	15	17	19	19	8
UTG edge	615	615	25	30	34	34	19
Activity coverage	3/31	3/31	2/28	2/28	2/28	2/28	2/28
Valid set text	0/70	12/70	0/10	0/20	0/9	0/9	2/21

Table 2: Coverage and effectiveness metrics for TFA Per-Login across all configurations.

5.3.2 Coverage and Effectiveness

Metric	D	DR	DL	DLC	H	HR	HL
UTG state	226	226	30	54	291	291	37
UTG edge	1112	1112	58	86	642	642	79
Activity coverage	3/31	3/31	2/28	2/28	2/28	2/28	2/28
Valid set text	48/101	39/101	8/19	37/93	102/210	30/210	27/28

Table 3: Coverage and effectiveness metrics for TFA Post-Login across all configurations.

Legenda: **DB**: DroidBot base, **DBR**: DroidBot + Replay output, **DBL**: DroidBot + LLAMA (text generation), **DLC**: DroidBot + LLAMA + Choice (decision making), **H**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLAMA

Omni-Notes Results

Key Findings

- All agent configurations (DroidBot base, DroidBot + LLaMA, DroidBot + LLaMA + Choice, Humanoid base, Humanoid + LLaMA) were tested.
- DroidBot base interacted with core app features: creating/editing/deleting notes, managing categories, reminders, permissions, and sharing notes.
- Adding LLaMA enabled richer actions: advanced categorization, search, voice dictation, and interaction with calendar activities.
- Humanoid agents performed more human-like actions: uploading files, checklist management, color changes, viewing note details, and password protection.
- Coverage analysis: DroidBot + LLaMA achieved the most states/transitions, while Humanoid agents covered more activities but explored less deeply.
- Integrating LLaMA for action choice improved functional coverage, but sometimes reduced state-space depth.
- Hybrid approaches (LLaMA + Humanoid) balance depth of exploration with variety and realism of interactions.

5.4.2 Coverage and Effectiveness

Metric	DB	DBR	DBL	DLC	H	HR	HL
UTG state	110	110	214	77	170	170	119
UTG edge	200	200	2087	232	360	360	230
Activity coverage	4/15	4/15	4/15	5/15	4/15	4/15	5/15
Valid set text	0/66	0/66	0/31	7/182	0/224	0/224	8/289

Table 2: Coverage and effectiveness metrics for Omni-notes across all configurations.

Legenda: **DB**: DroidBot base, **DR**: DroidBot + Replay output, **DL**: DroidBot + LLaMA (text generation), **DLS**: DroidBot + LLaMA + Choice (decision making), **HB**: Humanoid base, **HR**: Humanoid + Replay output, **HL**: Humanoid + LLaMA

Answers

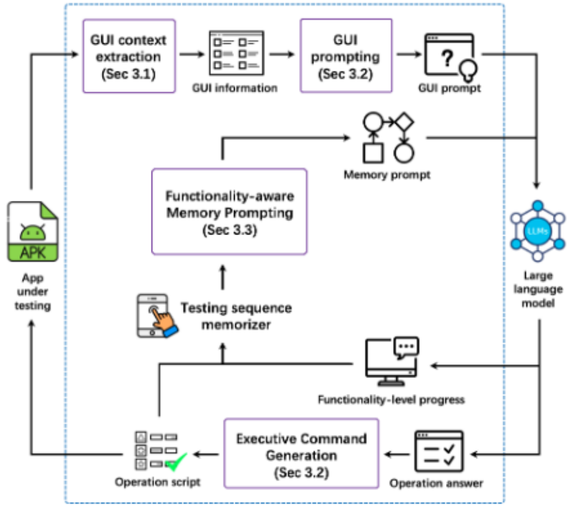
RQ1 - EFFECTIVENESS ANALYSIS

Experimental results show that LLMs can improve the generation of context-aware test cases, but their quantitative effectiveness varies greatly by application. While baseline methods (DroidBot, Humanoid) rarely generate valid textual inputs, LLM integration leads to modest improvements in complex apps and substantial gains when the application context is rich (e.g., 96.4% valid inputs for Humanoid + LLAMA in TFA Post-Login). Qualitatively, LLMs enable more sophisticated and human-like interactions, such as advanced field completion and the use of complex app features. However, practical limitations—hardware constraints, limited app descriptiveness, and a small app sample—currently mask the full theoretical potential of LLMs, which are inherently well-suited for semantic understanding and strategic reasoning in GUI testing

RQ2: COVERAGE OF TEST CASES

LLM-enhanced frameworks achieve greater diversity in explored paths, as shown by a higher number of state transitions (e.g., DroidBot + LLAMA reaches 2087 UTG edges). However, this does not always result in broader unique state or activity coverage, since many new paths are variations on existing flows. Notably, using LLMs for strategic decision-making (Choice) improves activity coverage (up to 33.3% vs. 26.7% baseline in PassAndroid and Omni-Notes), suggesting that LLMs can better navigate to previously inaccessible functionalities. Theoretically, LLMs offer strong potential for superior coverage by generating diverse, contextually relevant inputs and adapting exploration strategies based on app feedback

Identified Limitations and Challenges



LIMITATIONS

- **Hardware and computational resources** were restricted, limiting exploration and speed.
- **Test execution time was short**, reducing the number of states and transitions discovered.
- **App code and UI metadata were often poorly** descriptive, making it harder for LLMs to generate precise actions.
- **Sample size was small**, as only a few apps were tested, limiting the generalizability of the findings.

FUTURE IMPROVEMENTS

- **Use more powerful hardware** and allow longer test sessions for deeper exploration.
- **Promote better coding practices** and documentation to provide richer context for LLMs.
- Expand the evaluation to a larger and more **diverse set of apps**.
- **Develop advanced techniques for extracting and structuring context**, enabling LLMs to reason and select actions more effectively.

Discussion - LLM vs Manual Testing

LLM-based testing offers clear benefits over manual approaches, such as greater automation, efficiency, and the ability to generate diverse test cases without human effort. However, it does not fully solve the problem: test quality and coverage are still limited by technical and contextual constraints. While LLMs significantly reduce manual workload and promise better scalability as technology advances, further improvements and broader validation are needed before they can consistently replace manual test creation in every scenario. Integrating LLMs is a promising step forward, but not yet a complete solution.

Thanks for the attention

Q&A