



# Prompt Caching for Command-Driven IoT Systems

강 수 용

**Software Platforms Lab.**

Dept. of Computer Science, Hanyang University





# LLM-based Command-Driven IoT Systems

- Command-Driven IoT (Internet of Things) Systems
  - IoT systems that respond to user commands given in a spoken language
    - Responses to commands accompany **device controls**
  - Examples
    - Voice assistant systems: Apple Siri, Amazon Alexa, Samsung Bixby
    - Smart home devices
    - In-vehicle infotainment (IVI) systems

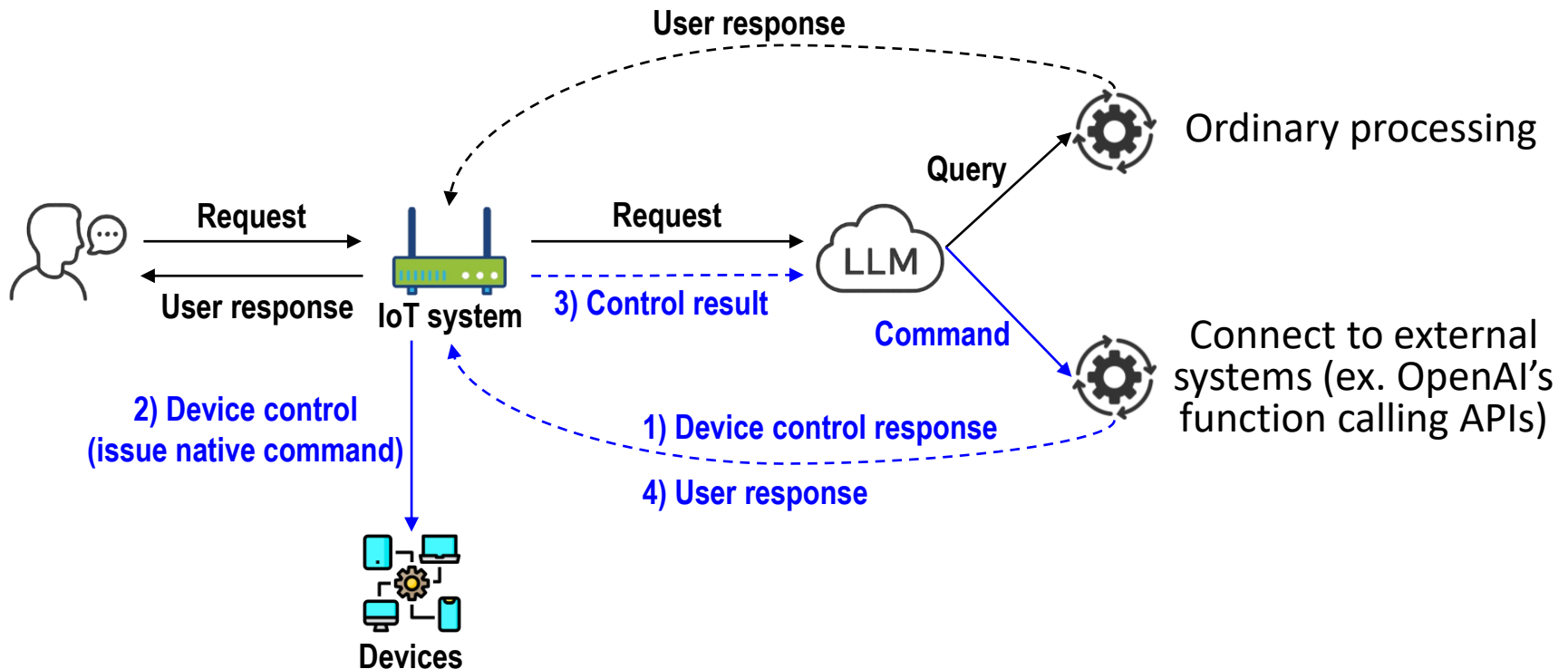


**Evolving to**

- LLM-based Command-Driven IoT Systems
  - Leverage external LLMs to
    - interpret natural language commands and
    - facilitate the execution of device control tasks
  - Enable the systems to conduct a natural dialogue with users or accurately control devices following user commands

# Request Processing in LLM-based Command-Driven Systems

- Request Types
  - **General queries**: lead to a natural dialogue with the system
  - **Control commands**: trigger device operations





## Problems with an LLM-based Command Processing

---

- Performance and Economy
  - Even a simple command, such as “Turn on the light in the kitchen”, incurs a large response time and cost
- Accuracy
  - Due to the intrinsic characteristics of generative AI models, LLMs may respond differently to identical commands → very harmful to device control commands
- Requirements
  - Optimization schemes that reduce response time and operating costs while maintaining consistency in responses
  - **Prompt Caching**: an effective optimization strategy for LLMs



## Problems with Legacy Prompt Caching

- Diversity of command expressions
  - **Action expression** varies depending on user's different expressions
    - “turn on” = “switch on” = “power on” = ...
  - An action expression can be applied to different target devices
    - ‘radio’, ‘light’, ‘air conditioner’, ...
  - Each device may have various expressions
    - TV = television = telly = ...
- Problems of Legacy Prompt Caching for IoT systems
  - Keeps (input, output) pairs of the processed prompts in the cache
  - Inherits an intrinsic limitation of ordinary caching schemes
    - Recognizes slightly different expressions as entirely distinct prompts → Low cache hit ratio
    - Particularly harmful to command-driven systems where users frequently issue identical commands with different arguments
      - “Set the temperature to 25 degrees”
      - “Set the temperature to 26 degrees”



## Our Approach: Snippet Caching Strategy



# Snippet Caching Strategy

- Basic Idea
  - A command can be perceived as a function call with specified arguments
    - “Turn on the light in the kitchen” : `turn_on(light, kitchen)`
    - “Turn on the radio in the bedroom” : `turn_on(radio, bedroom)`
  - Maintain a cache entry for each function
    - Process all commands that invoke cached functions, *locally* without LLM calls
- Requirements to realize the idea
  - Controls and responses to commands w/ different arguments must be different
    - “The light in the kitchen has been turned on”
    - “The radio in the bedroom has been turned on”
  - Solution: Cache two customized code snippets for each function
    - One for arguments extraction and another for response generation,
    - instead of caching legacy (input, output) pairs



# Handling Expression Diversity in the Snippet Caching Strategy

- Snippet Caching maintains
  - A set of ‘action types’ w/ respective ‘action parameters’
    - **Action type**: pre-determined general term, representatively expressing an action
      - Ex. ‘activate’ : representative expression for “turn on”, “switch on”, “power on”...
    - **Action parameters**: specific conditions required to take an action
      - Ex. ‘activate’ requires the target **object** and (optionally) **location** of the object
  - Thesaurus for each action type and action parameter
  - Two Code Snippets for each Action Type
    - **Argument Extractor**: extracts action parameter values from user commands
    - **Response Generator**: generates LLM-like responses to user commands
    - Snippets mimic LLM operations for the action type



## Command Processing in the Snippet Caching Strategy

- **Example Command: “Turn on the lamp in the kitchen”**
  - 1) Identifies the action type of the command
    - Lookups the cache to find an action type to which the action expression (“turn on”) belongs
    - Cache hit: a cache entry for the ‘activate’ action type is present in the cache
  - 2) Extracts arguments for the identified action type
    - Extract values of object (‘lamp’) and location (‘kitchen’) parameters for the ‘activate’ action type
    - Using the argument extractor in the cache entry
  - 3) Issues a native command, based on the action type and arguments
  - 4) Generates a customized user response
    - Pass the arguments and control result to the response generator
    - User response: “The lamp in the kitchen has been turned on”



## Effectiveness of the Snippet Caching Strategy

- Performance and Economy
  - Maintains only one cache entry for each action type
  - Functionally equivalent (i.e., same action type) commands are processed using a single cache entry
    - High cache hit ratio → significantly reduces response times and operation costs
- Consistent responses to commands with the same action type
  - The response generator creates identical responses (with different arguments) to all commands of an action type
  - Examples for the 'activate' action type
    - Command: "Switch on the **lamp** in the **kitchen**"
    - Response: "The **lamp** in the **kitchen** has been turned on"
    - Command: "Turn on the **radio** in the **bedroom**"
    - Response: "The **radio** in the **bedroom** has been turned on"

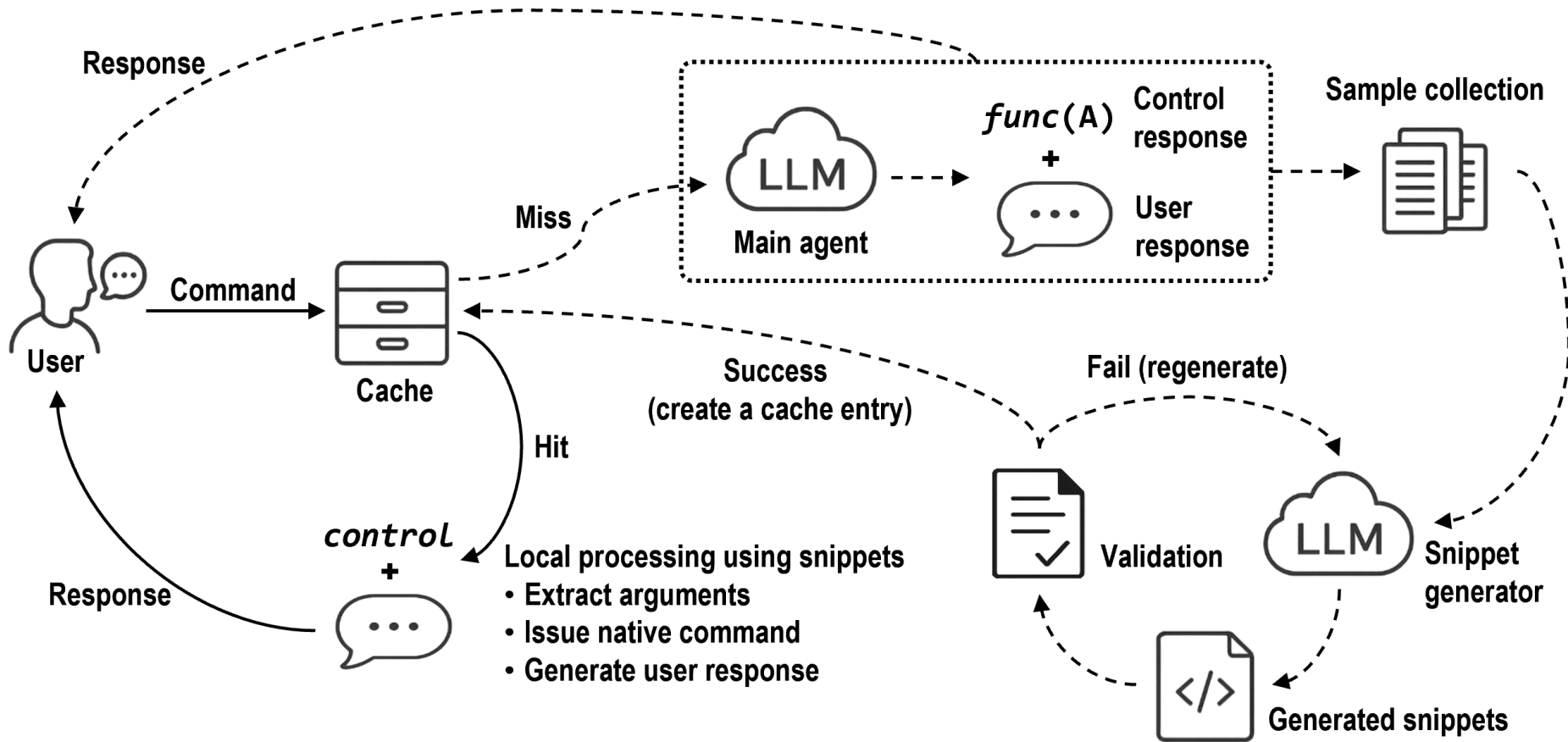


## Snip-Cache: A Realization of the Snippet Caching Strategy



- Snip-Cache
  - Exploits the powerful NLP & code generation capabilities of modern LLMs
  - Uses two kinds of external LLMs
    - **Main agent**: Command processing LLM
    - **Snippet generator**: Cache component generation LLM
- Key Features
  - LLM-based dynamic cache component generation
    - Dictionaries (Thesaurus)
      - **Action Trigger Dictionary (ATD)** : handles diverse action expressions
      - **Action Argument Dictionary (AAD)** : handles diverse argument expressions
      - Issues “**Dictionary Generation Prompt**” to the snippet generator
    - Snippets
      - For the argument extractor, issues “**Argument Extraction Snippet Prompt**”
      - For the response generator, issues “**Reponse Generator Snippet Prompt**”
    - Embeds LLM responses in the prompts as samples → **Snip-Cache mimics LLM operations**
  - Validation tests on the generated components

# Snip-Cache: Overall Command Processing Flow



## Snip-Cache: Action Type Specification (ATS)

- A command-driven system has limited controllable features
  - System developers can define applicable action types in advance for use in both the system and external LLMs
  - **Action Type Specification (ATS)**: definition of an action type, including the action type name and required action parameters for the type

```
{
  "name": "activate",
  "description": "Activates a specified object, such as a TV, heater, or lamp. If applicable, the action is performed at the specified location (e.g., bedroom, kitchen, or washroom).",
  "parameters": {
    "type": "object",
    "properties": {
      "object_name": {
        "type": "string",
        "description": "Target object to activate, one of 'lamp', 'TV', or 'heater'."
      },
      "location_name": {
        "type": "string",
        "description": "Optional location where the action should be performed, such as 'bedroom', 'kitchen', or 'washroom'."
      }
    }
  },
  "required": ["object_name"],
  "additionalProperties": false
}
```

Native command (device control function) name

Representative expressions used in the native command

An example ATS for the 'activate' action type



## Snip-Cache: Main Agent

---

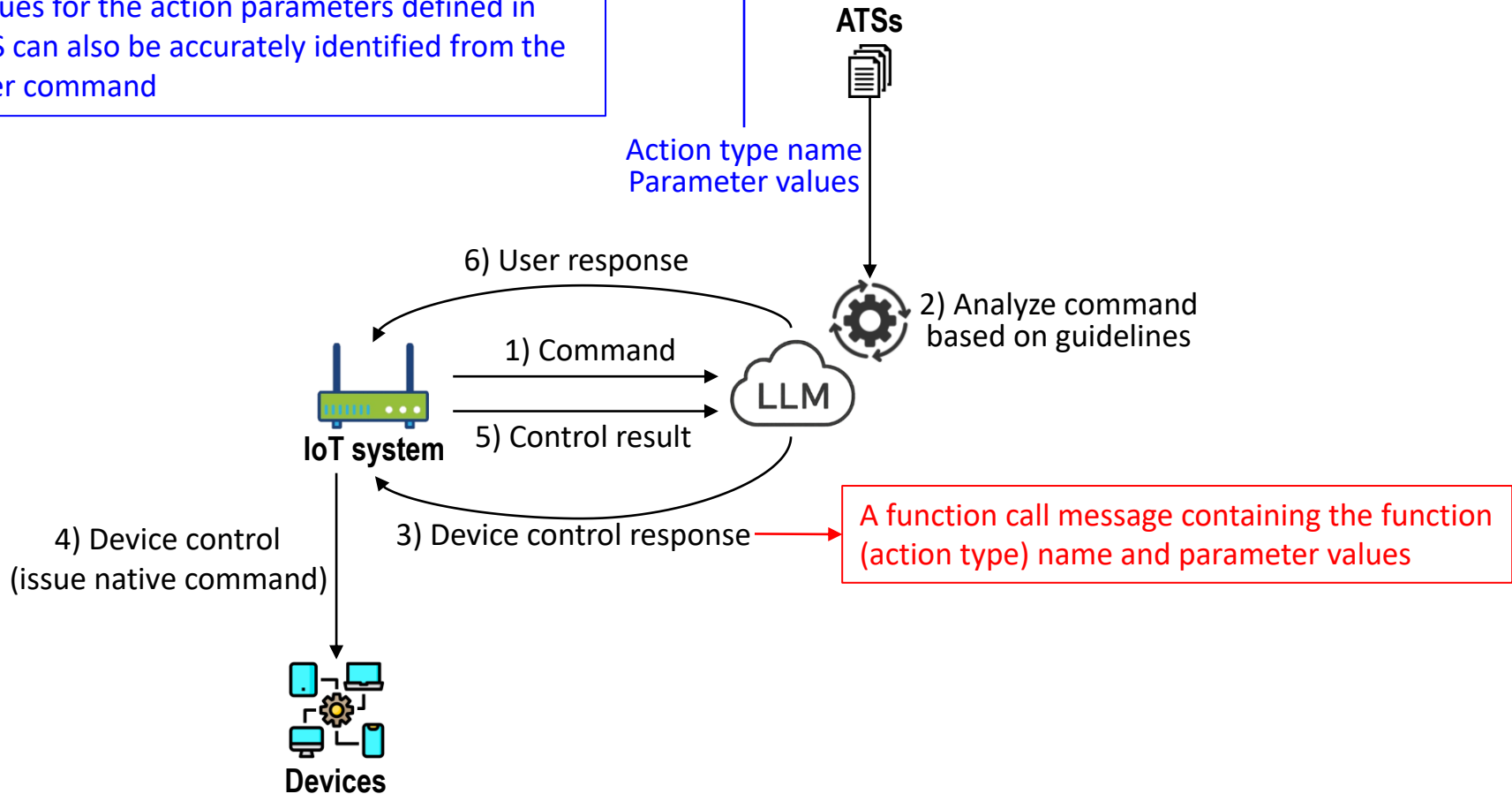
- Making the main agent ready for command processing
  - **Device control functions** (i.e., native commands) are registered to the main agent, in advance, using ATSs
    - The current structure of an ATS follows the function schema in the ‘tools’ parameter of OpenAI APIs to use OpenAI LLM

- **Command processing guidelines** (Instruction prompts)

Example) *“You are a lighting system control agent that can turn on or off a light in an arbitrary location. The location can be a bedroom, dining room, or kitchen. You can call the activate or deactivate function when you turn on or off a light, respectively.”*

# Snip-Cache: Interactions with the Main Agent

- Different user expressions for an action can be accurately interpreted as the corresponding action type
- Values for the action parameters defined in ATS can also be accurately identified from the user command





# Snip-Cache: Interactions with the Main Agent

- LLM Interaction Sample

- Leaves the interactions with the main agent in an LLM interaction sample
  - Groups the collected samples by action type
- The samples provide examples of
  - Different action expressions for each action type
  - Extracted values for each action parameter
  - LLM-style user responses
- Snip-Cache provides samples to LLM for a cache entry generation

| Field                           | Example               | Generated by |
|---------------------------------|-----------------------|--------------|
| User Command                    | Turn on the lights    | User         |
| Action type Name                | activate              | Main Agent   |
| Action parameter values         | {object_name: lights} | Main Agent   |
| Native command execution result | True                  | System       |
| Final response                  | Lights are now on     | Main Agent   |

# Snip-Cache: Cache Entry Creation: Dictionary Generation

## An example of the 'Dictionary Generation prompt'

Generate semantic dictionaries for the action type `activate`, based on the following user command examples and Action Type Specification.

### [User Command Examples]

- ```
# 1: "Turn on the TV in the bedroom"
      → activate (object_name: TV, Location_name: bedroom)

# 2: "Switch on the heater"
      → activate (object_name: heater)

# 3: "Power on the Lamp in the washroom"
      → activate (object_name: Lamp, Location_name: washroom)
```

### [Specification and Constraints]

#### Action Type Specification (ATS)

Dynamically filled when issuing a real prompt

### [Requirements]

Based on this example, perform the following tasks:

1. Create an **Action Trigger Dictionary (ATD)**:  
Extract the core phrase(s) that indicate the user's intention to trigger.  
Generate natural variations of the core phrase for the same action.  
...
2. Create an **Action Argument Dictionary (AAD)**:  
For each parameter name in ATS, list representative expressions that appear in its description field.  
Add varied expressions of each representative expression by imagining alternative user commands.  
...

### [Note]

Parameters, like "date", are skipped if no representative value is defined in the ATS.



Snippet generator

### [Output]

```
"ATD": [
  "activate",
  "turn on",
  "switch on",
  "start",
  "power on",
  "enable",
  ...
]

"AAD": {
  "object_name": {
    "lamp": [
      "lamp",
      "lighting device",
      "lights",
      "lamps",
      ...
    ],
    "TV": [
      "TV",
      "television",
      "telly",
      ...
    ],
    ...
  }
}
```

# Snip-Cache: Cache Entry Creation: Snippet Generation

Generate a Python function `'extract_arguments(command: str, argument_dictionary: dict) → dict'` that extracts argument values from a user command using the action type specification and argument mappings.

## [Instructions]

- Match enum-type arguments using `argument_dictionary`.
- Use `word2number` or `dateparser` for numbers and dates.
- Do not match substrings or partial tokens.
- Always return all required arguments; use `None` if not found.

## [Matching Rules]

- Normalize input (`lowercase`, `strip`, `tokenize`).
- Avoid generic one-word triggers like `please`, `on`, `off`, or `more`.
- Match multi-word expressions using exact phrase match.
- Use regular expressions only for numeric/date types.

## [Function Output Example]

```
{"object_name": "TV", "location_name": "bedroom"}
```

## [Allowed Libraries]

`word2number`, `dateparser`, `re`

## [Context]

- Function Specification and Constraints:

*Action Type Specification (ATS) : Activate*

- Example User Commands:

```
# 1: "Turn on the TV in the bedroom"
→ activate (object_name: TV, Location_name: bedroom)

# 2: "Switch on the heater"
→ activate (object_name: heater)

# 3: "Power on the Lamp in the washroom"
→ activate (object_name: Lamp, Location_name: washroom)
```

- Argument Mapping Dictionary (AAD):

```
"object_name": {
  "Lamp": ["Lamps", "lighting device", ...],
  "tv": ["tv", "television", ... ],
  ...
},
"Location_name": {
  "kitchen": ["kitchen", "cooking room", ...],
  ...
}, ...
```

An example of the 'Argument Extraction Snippet prompt'

ATS informs of the action parameters whose values will be extracted.

## [Output]

```
def extract_arguments(
    command: str,
    argument_dictionary: dict)
    -> dict:
    ...
    return extracted_arguments
```

Samples provide examples of extracted values for each action parameter.

AAD informs of the internal format of the dictionary, so that the generated snippet can correctly parse and use the input dictionary.



# Snip-Cache: Cache Entry Creation: Snippet Generation

Generate a Python function '**generate\_response(arguments: dict, action\_result: bool) → str**' that returns a natural-sounding English sentence based on extracted arguments and action outcome.

## [Instructions]

- Use all provided arguments; handle missing ones gracefully.
- Reflect both the argument values and whether the action succeeded.
- Always return a single, well-formed, spoken-style sentence.
- Do not raise exceptions or include comments/logs.

## [Style and Generalization]

- Match the tone and structure of sample user responses.
- Avoid hardcoding values; make the sentence reusable.
- Use general fallback messages for incomplete or failed actions.

## [Scenarios to Handle]

- Success with complete arguments → Confirmation message
- Success with partial arguments → Acknowledge missing info
- Failure → Polite fallback or error message

## [Context]

- Function Specification and Constraints :

*Action Type Specification (ATS) : Activate*

- User Command Examples and Expected Responses

```
# 1: "Turn on the TV in the bedroom"
→ arguments: {object_name: TV, Location_name: bedroom}
→ action_result: True
→ response: "The TV in the bedroom has been turned on."

# 2: "Switch on the heater"
→ arguments: {object_name: heater}
→ action_result: True
→ response: "The heater has been turned on."

# 3: "Power on the Lamp in the washroom"
→ arguments: {object_name: Lamp, Location_name: washroom}
→ action_result: False
→ response: "Failed to turn on the Lamp in the washroom."
```

An example of the 'Response Generator Snippet prompt'



Snippet generator

## [Output]

```
def generate_response(
    arguments: dict,
    action_result: bool) -> str :
    ...
    return response
```

ATS guides the snippet generator to produce a customized snippet to the action type.

Responses in the samples provide examples of LLM-style user responses when parameter values and native command execution result are given.



## Snip-Cache: Cache Entry Creation: Structure of an Entry

- A new cache entry is generated when a predefined number of LLM interaction samples are collected for an action type

activate

**Action Trigger Dictionary (ATD):** activate, turn on, switch on, power on, enable, initiate, ...

**Action Argument Dictionary (AAD):**

```
object_name:    {lamp : [lamp, lamps, illumination, ...]},  
                {music : [music, audio, songs, ...]},  
                ...
```

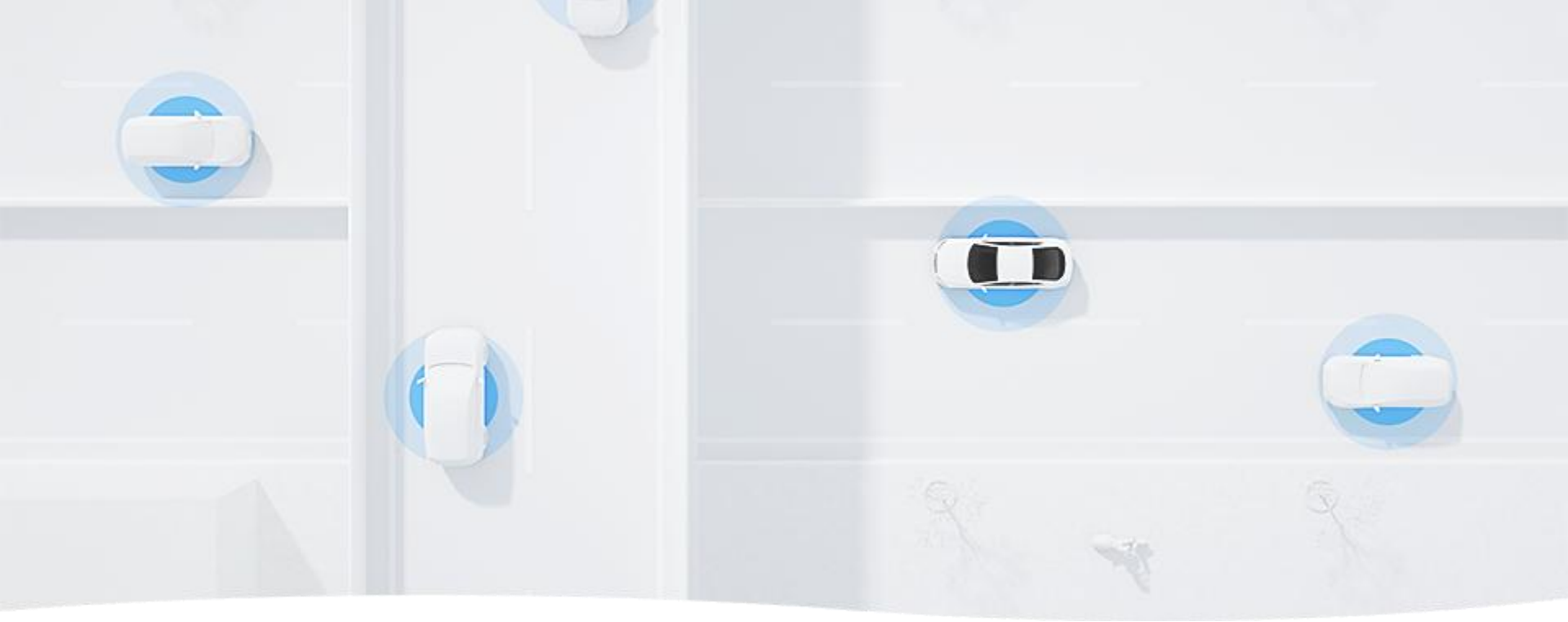
```
location_name : {kitchen: [kitchen, cooking area, dining area, ...]},  
                {washroom: [bathroom, restroom, washroom, ...]},  
                ...
```

**Argument extractor snippet:**

```
def extract_arguments(command: str, argument_dictionary : dict) -> dict:  
    ...  
    return extracted_arguments
```

**Response generater snippet:**

```
def generate_response(arguments: dict, action_result: bool) -> str:  
    ...  
    return response
```



## Performance Evaluation and Conclusion



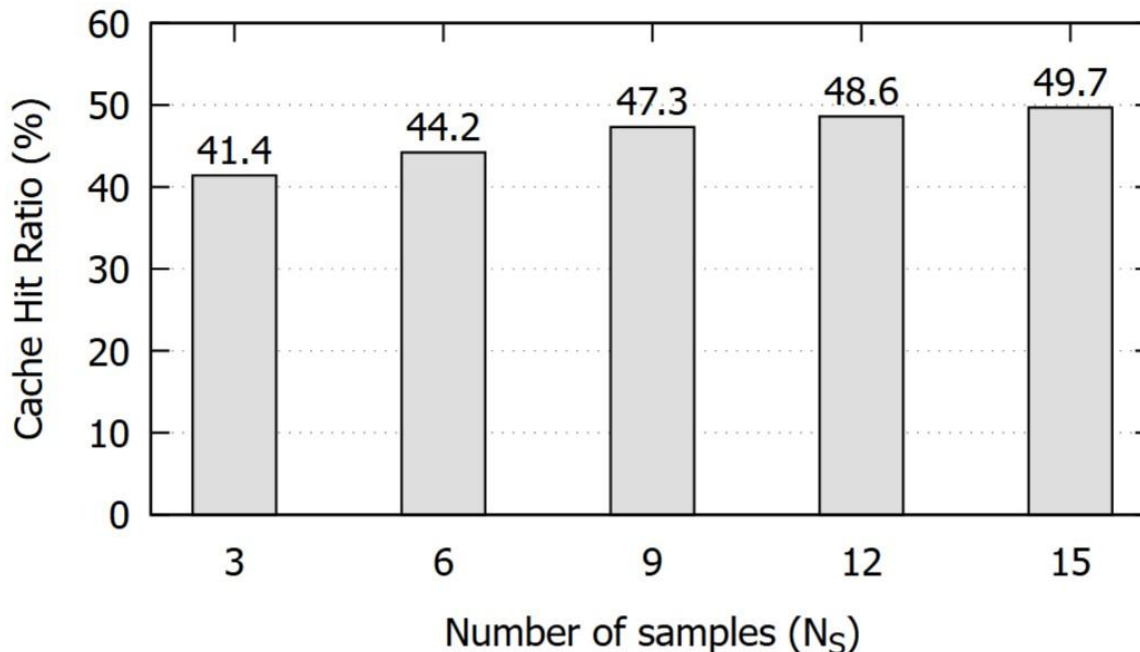
# Experimental Environment

---

- Virtual command-driven system
  - Accepts user commands in written (not spoken) natural language
  - Outputs a user response and a native device control command
  - Runs on Windows 11 Pro 64-bit system w/ AMD Ryzen 7 5700X (8 core), 32 GB RAM, and a 512 GB NVMe SSD
- Snip-Cache
  - Uses ChatGPT API to exploit OpenAI's gpt-4o LLM
  - Used Azure OpenAI LLM service in the East US 2 region for both the main agent and the snippet generator
- GPTCache: A competing (semantic) caching scheme
  - Used its codebase in the LangChain framework
  - Vector embedding model: a lightweight paraphrase-albert-small-v2 model in the Sentence-Transformers framework
- Workload: Fluent Speech Commands dataset
  - Consists of user commands for smart home appliances or virtual assistants (total 6 action types)
  - Removed duplicated commands (total 248 different ones), and augmented them using an LLM (total 1,200 commands)

## Experimental Results: Cache Hit Ratio

- Unique characteristics of Snip-Cache
  - Cache size independence
    - Cache replacement policy is not required
  - Number of collected samples ( $N_s$ ) affects the cache hit ratio
- Among 1,200 commands, 496 ( $N_s=3$ ), 530 ( $N_s=6$ ), 568 ( $N_s=9$ ), 583 ( $N_s=12$ ), 596 ( $N_s=15$ ) commands are processed by the Snip-Cache



### Classification of cache misses

- ATD matching failed: 89.28%
- Multiple matching ATD found: 0.67%
- NULL value extracted for mandatory parameters: 10.05%



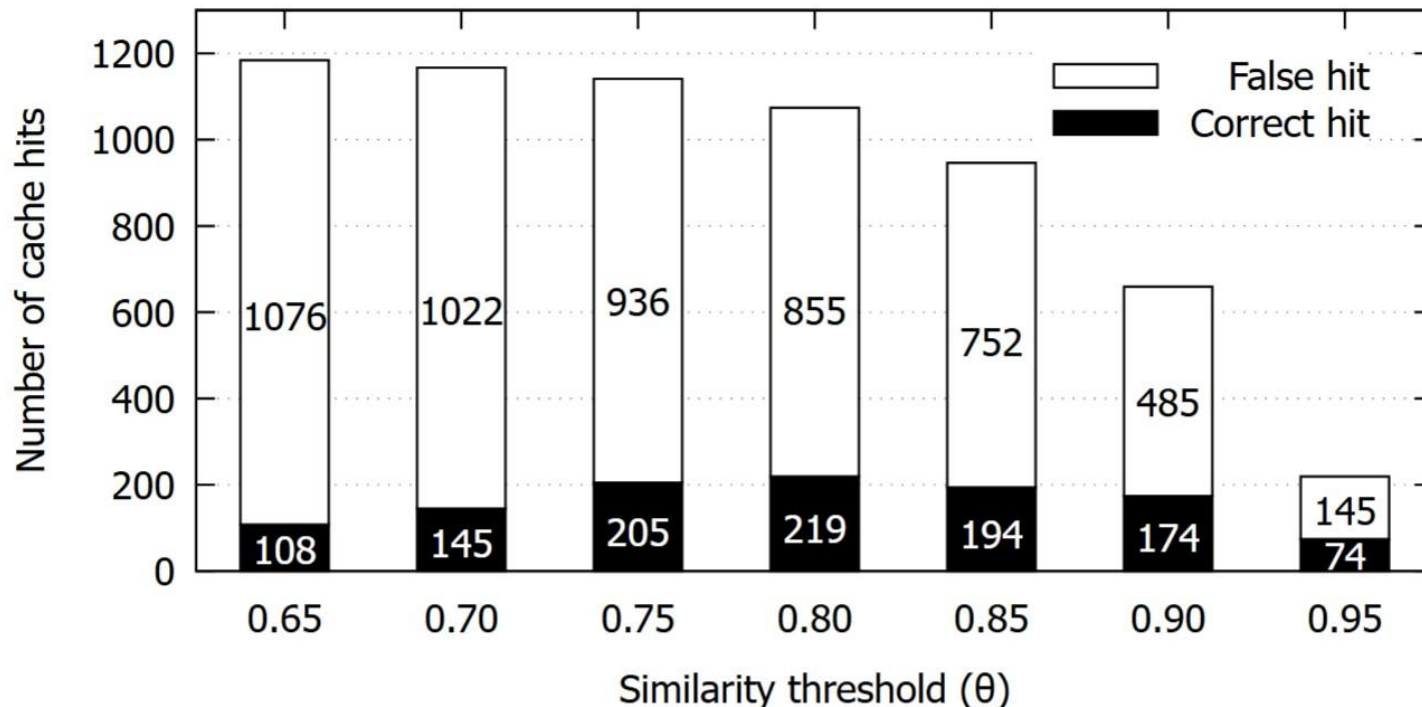
# Experimental Results: Response Accuracy

- Evaluation Method
  - LLM-generated responses are deemed to be correct
  - Compared the cache-generated native commands with the LLM-generated ones
    - If identical, the cache-generated responses are assumed to be correct
  - Accuracy = # of correct responses / # of total commands (= 1,200)
    - Correct responses include the LLM-generated responses upon cache misses
- Accuracy of the Snip-Cache
  - 99.92% (only one incorrect one among 1,200 responses)

|                                |                                                                                                                                     |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| User Command                   | “Tone down the audio in the kitchen.”                                                                                               |
| LLM-generated native command   | decrease(volume, kitchen)                                                                                                           |
| Cache-generated native command | deactivate(music, kitchen)                                                                                                          |
| Cause of inaccuracy            | The term ‘tone down’ is missing in the ATD of the ‘decrease’ action type but is present in the ATD of the ‘deactivate’ action type. |

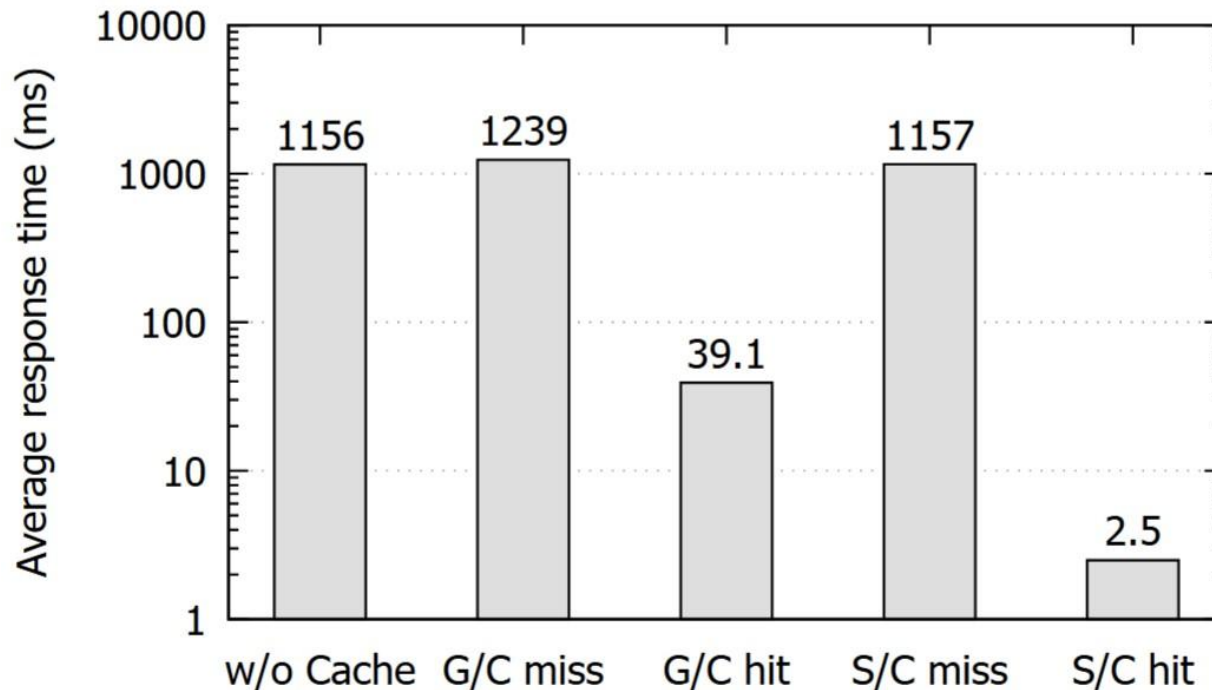
## Experimental Results: Response Accuracy

- Accuracy of GPTCache varies with the similarity threshold ( $\theta$ )
  - With  $\theta=0.65$ , accuracy = 10.3% and hit ratio = 98.6%
  - With  $\theta=0.95$ , accuracy = 87.9% and hit ratio = 18.3%
  - We use  $\theta=0.8$  with which GPTCache showed the largest number of correct hits



## Experimental Results: Response Time

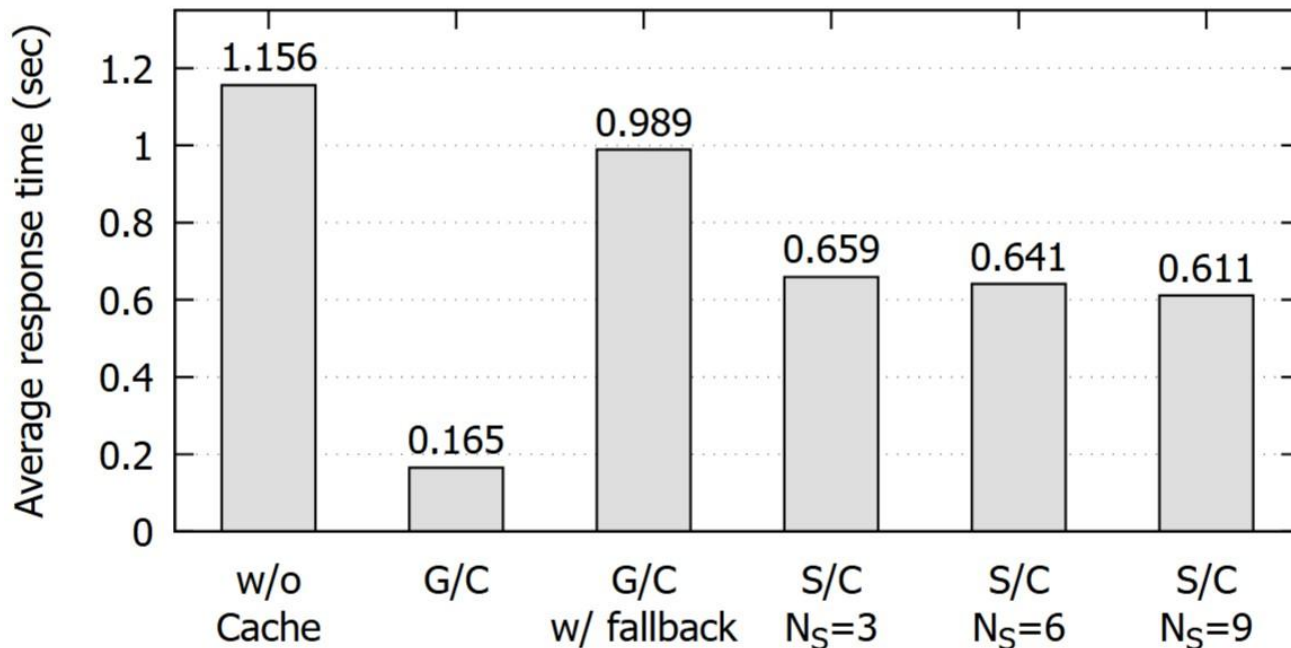
- Response Time
  - Elapsed time from the command input time to the time when
    - the command-driven system either generated a user response (cache hit)
    - or received it from the main agent (cache miss)
  - GPTCache showed larger cache lookup time than Snip-Cache due to the complex semantic similarity comparison between an input command and every cached command → G/C cases > S/C cases



G/C: GPTCache  
S/C: Snip-Cache

## Experimental Results: Response Time

- Average Response Time
  - Snip-Cache showed 43% ~ 47% better response time, compared with the w/o Cache case
  - Significantly lower response time of G/C is due mainly to the false hits
    - With a fallback mechanism (using an LLM when a false hit occurs) applied, G/C shows far worse response time than S/C

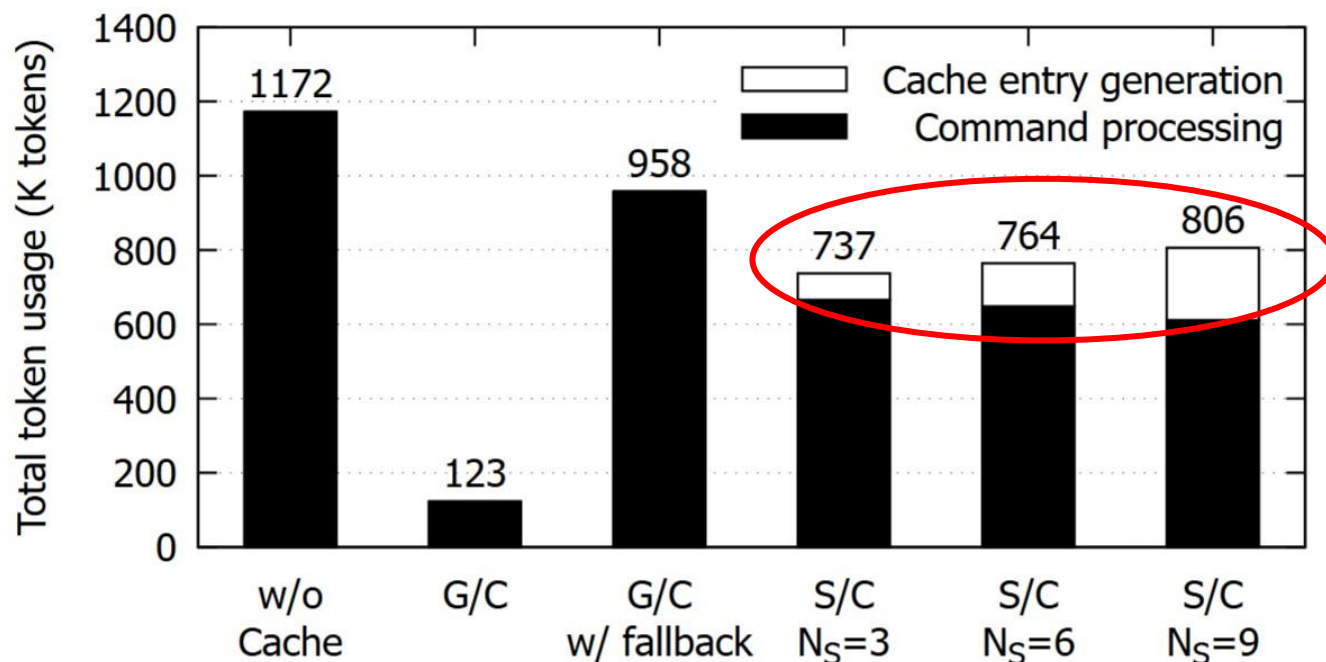


## Experimental Results: Token Usage

- Pros and Cons of Snip-Cache in terms of the token usage
  - Reduces token usage by processing a portion of commands locally
  - Uses additional tokens for generating and validating cache entries

Additional token usage in S/C

| $N_S$ | Total token usage | Average token usage per action type | Break-even point per action type |
|-------|-------------------|-------------------------------------|----------------------------------|
| 3     | 70,500            | 11,750                              | $\approx 12$ hits                |
| 6     | 116,244           | 19,374                              | $\approx 20$ hits                |
| 9     | 194,604           | 32,434                              | $\approx 33$ hits                |

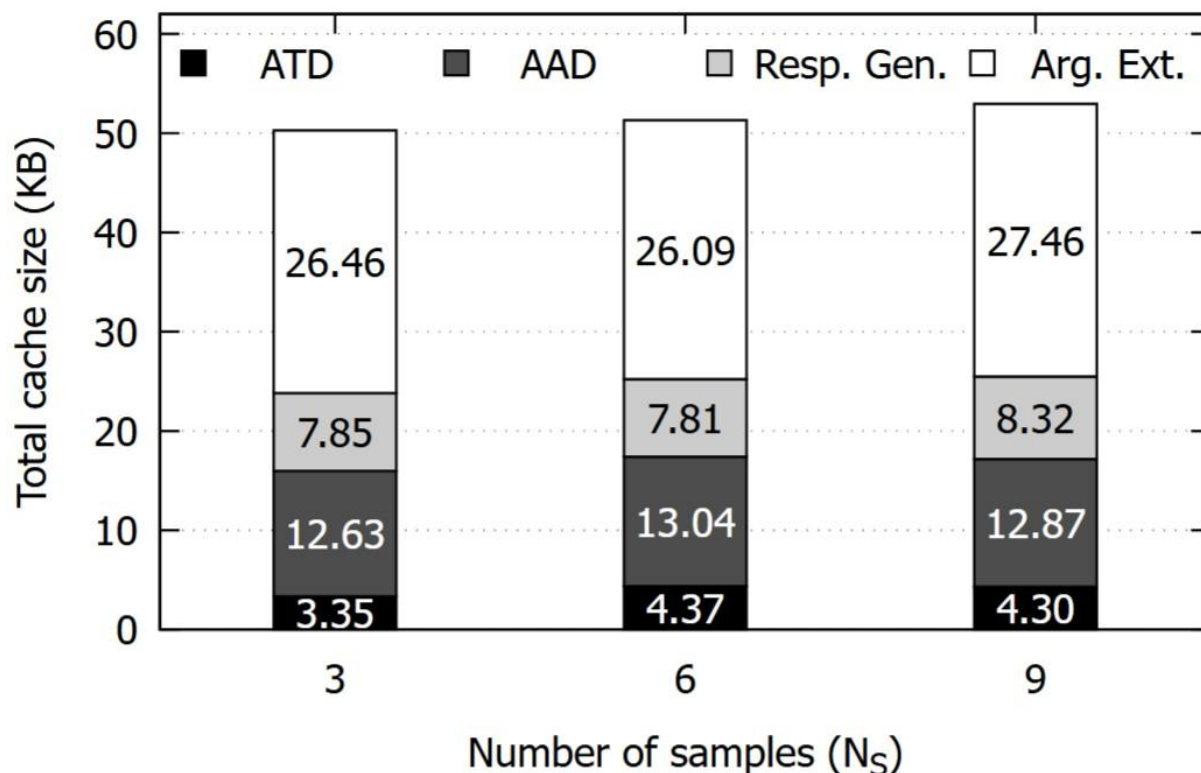


Increases due mainly to more frequent supplementary prompt issues as  $N_S$  grows.

Average validation failures:  
 $N_S=3 \rightarrow 4.8$  times  
 $N_S=6 \rightarrow 11.6$  times  
 $N_S=9 \rightarrow 15.4$  times

## Experimental Results: Space Consumption

- GPTCache ( $\theta=0.8$ ) consumes
  - 1.07 MB of space for a database file (sqlite.db) that stores cache entries
  - 46.5 MB of space for the vector embedding model
- Snip-Cache consumes negligible amount of space ( $< 60$  KB)





## Conclusion

- Characteristics of Snip-Cache
  - Perceives user commands as function calls
  - Maintains two code snippets that mimic LLM operations for each function
  - Enables functionally equivalent commands to be processed locally by a single cache entry, reducing reliance on external LLMs
- Performance of Snip-Cache
  - Significantly reduces response time and token usage with negligible amount of cache space
  - Maintains high response accuracy
- Conclusion
  - Snip-Cache can be a practical solution for LLM-based command-driven IoT systems
  - Offers a new avenue in designing prompt caching schemes for LLMs