POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Toward Context-aware LLM-driven UAVs

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Christian Mariano**

Student ID: 260338
Advisor: Prof. Luca Mottola
Academic Year: 2025-26

Abstract

Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, are rapidly transitioning from expert-operated platforms to autonomous systems accessible to non-specialist users. This evolution raises new challenges for Human-Drone Interaction (HDI), a research area situated within the broader field of Human-Robot Interaction (HRI), where interaction is increasingly understood as a cognitive and social process shaped by perception, context, and mutual adaptation. While early HDI systems focused on manual piloting or gesture-based control, recent advances in Large Language Models (LLMs) and multimodal reasoning have enabled drones to interpret natural-language goals and autonomously generate executable plans. Despite these advances, current approaches remain split between interaction paradigms that support user-driven customization of the interface and systems designed mainly for goal-oriented communication.

Writable interaction paradigms, exemplified by systems such as `DronWO` [13], allow users to actively shape drone behavior by defining new executable actions through gesture-based sequences. These approaches promote expressive and customizable interaction but often impose cognitive overhead and lack mechanisms for contextual reasoning and long-term adaptation. Conversely, LLM-based frameworks such as `TypeFly` [19] reduce interaction complexity by enabling users to describe goals directly in natural language, delegating planning and execution to the system. Although this paradigm supports flexible task decomposition, it provides limited support for user-driven extension, environmental awareness, or learning from past interactions.

This thesis addresses the gap between these directions by proposing a context-aware HDI framework that integrates writable interaction with reasoning-based autonomy. We introduce `Drone Adaptive Context-aware System` (`DACS`), an adaptive architecture that turns natural-language interaction into a continuous co-learning process: the drone learns user preferences and environmental structure, while users progressively learn how to guide and personalize the system through interaction. The system is organized around a conceptual model in which an LLM acts as the central reasoning component, mediating interaction between the user, a multi-level memory structure, environmental perception, and the drone's physical execution. The system supports adaptation at both the interaction

and environmental levels, allowing the drone to interpret user intent, learn from previous experiences, and reason about the surrounding context when planning its behavior. Interaction progressively evolves as the drone refines its responses through experience while users simultaneously learn how to express goals more effectively. This mutual adaptation shifts human-drone interaction from isolated command execution toward a continuous and context-aware form of collaboration.

To validate the proposed approach, we conducted a comparative user study investigating how interaction evolves when adaptive capabilities are introduced into HDI. Ten participants performed open-ended tasks, including object search and repeated requests, allowing observation of how users formulate commands, provide feedback, and progressively shape system behavior over time. Results show that interaction with the baseline `TypeFly` frequently stalled when targets were not immediately visible. In these situations, users manually intervened by restarting execution and issuing sequences of low-level commands (e.g., *"move forward by..."*, *"turn by..."*) to locate hidden objects, as the baseline lacks an autonomous exploration strategy. Questionnaire responses reflect this limitation: four out of five baseline participants reported little or no perception that the drone remembered previously visited regions, and confidence in exploration behavior remained low. Because each request is handled in isolation, interaction with the baseline was often perceived as fragmented and reactive rather than collaborative.

In contrast, interaction with `DACS` evolved as a continuous exchange in which both the drone and the user adapted over time. The system interpreted feedback as guidance influencing subsequent behavior, reused past observations to inform new decisions, and adjusted exploration strategies based on the surrounding context. All participants interacting with the adaptive system reported high confidence in exploration behavior, with ratings of 4-5 on a 5-point Likert scale (1 = very low, 5 = very high), together with consistently high perceived usefulness of feedback. Participants described the interaction as clearer and less frustrating, highlighting a stronger sense of cooperation and indicating that context-aware adaptation transforms HDI from isolated command execution into an evolving collaborative process.

Keywords: Human-Drone Interaction (HDI), Human-Robot Interaction (HRI), Large Language Models (LLMs), Adaptive Interfaces, Context-Aware Systems.

Abstract in lingua italiana

I veicoli aerei senza pilota (UAV), comunemente noti come droni, stanno rapidamente evolvendo da piattaforme destinate a operatori esperti a sistemi autonomi accessibili anche a utenti non specialisti. Questa transizione introduce nuove sfide per l'Interazione Uomo-Drone (HDI), ambito di ricerca collocato nel più ampio campo dell'Interazione Uomo-Robot (HRI), in cui l'interazione è intesa come un processo cognitivo e sociale influenzato da percezione, contesto e adattamento reciproco. Se i primi sistemi di HDI si basavano prevalentemente sul pilotaggio manuale o su controlli gestuali, i recenti progressi nei Modelli Linguistici di Grandi Dimensioni (LLM) e nel ragionamento multimodale consentono ai droni di interpretare obiettivi espressi in linguaggio naturale e di generare autonomamente piani eseguibili. Nonostante tali avanzamenti, gli approcci attuali risultano ancora polarizzati tra paradigmi orientati alla personalizzazione dell'interazione e sistemi progettati principalmente per una comunicazione focalizzata sull'obiettivo.

I paradigmi di interazione "scrivibile", come il sistema `DronWO` [13], permettono agli utenti di modellare attivamente il comportamento del drone definendo nuove azioni eseguibili attraverso sequenze gestuali. Sebbene tali approcci favoriscano interazioni espressive e altamente personalizzabili, introducono un significativo carico cognitivo e presentano limitazioni in termini di ragionamento contestuale e adattamento a lungo termine. Al contrario, framework basati su modelli linguistici, come `TypeFly` [19], riducono la complessità dell'interazione consentendo agli utenti di descrivere gli obiettivi direttamente in linguaggio naturale, delegando al sistema le fasi di pianificazione ed esecuzione. Tuttavia, questo paradigma offre un supporto limitato all'estensione guidata dall'utente, alla consapevolezza ambientale e all'apprendimento derivante dalle interazioni precedenti.

Questa tesi affronta il divario tra tali approcci proponendo un framework di HDI sensibile al contesto che integra interazione scrivibile e autonomia basata sul ragionamento. Introduciamo il `Drone Adaptive Context-aware System` (`DACS`), un'architettura adattiva che trasforma l'interazione in linguaggio naturale in un processo continuo di co-apprendimento: il drone apprende le preferenze dell'utente e la struttura dell'ambiente, mentre gli utenti imparano progressivamente a guidare e personalizzare il sistema attraverso l'esperienza interattiva. L'architettura è organizzata attorno a un modello con-

cettuale in cui un modello linguistico agisce come componente centrale di ragionamento, mediando l'interazione tra utente, una struttura di memoria multilivello, la percezione ambientale e l'esecuzione fisica del drone. Il sistema supporta l'adattamento sia a livello interattivo sia a livello ambientale, consentendo al drone di interpretare l'intento dell'utente, apprendere dalle esperienze passate e ragionare sul contesto circostante durante la pianificazione del comportamento. L'interazione evolve progressivamente: il drone affina le proprie risposte attraverso l'esperienza, mentre gli utenti apprendono a formulare gli obiettivi in modo sempre più efficace. Tale adattamento reciproco trasforma l'HDI da una sequenza di comandi isolati a una forma continua e contestualmente consapevole di collaborazione.

Per validare l'approccio proposto è stato condotto uno studio comparativo con utenti, volto ad analizzare come l'interazione evolva con l'introduzione di capacità adattive. Dieci partecipanti hanno svolto attività aperte, tra cui la ricerca di oggetti e richieste ripetute, consentendo di osservare come gli utenti formulino i comandi, forniscano feedback e modellino progressivamente il comportamento del sistema nel tempo. I risultati mostrano che l'interazione con il sistema baseline `TypeFly` si interrompeva frequentemente quando l'oggetto target non era immediatamente visibile. In tali situazioni, gli utenti intervenivano manualmente riavviando l'esecuzione e impartendo sequenze di comandi a basso livello (ad esempio *"avanza di..."*, *"ruota di..."*) per individuare oggetti nascosti, poiché il sistema baseline non dispone di una strategia autonoma di esplorazione. Le risposte ai questionari riflettono tale limitazione: quattro partecipanti su cinque nel gruppo baseline hanno riportato una scarsa o nulla percezione della capacità del drone di ricordare le regioni precedentemente visitate, e la fiducia nei comportamenti di esplorazione è risultata bassa. Poiché ogni richiesta viene gestita in modo indipendente, l'interazione con la baseline è stata spesso percepita come frammentata e reattiva, piuttosto che collaborativa.

Al contrario, l'interazione con `DACS` si è sviluppata come uno scambio continuo in cui sia il drone sia l'utente si adattavano nel tempo. Il sistema interpretava il feedback come guida per i comportamenti successivi, riutilizzava osservazioni pregresse per informare nuove decisioni e adattava le strategie di esplorazione in funzione del contesto circostante. Tutti i partecipanti che hanno interagito con il sistema adattivo hanno riportato un elevato livello di fiducia nei comportamenti di esplorazione, con valutazioni pari a 4–5 su una scala Likert a 5 punti (1 = molto bassa, 5 = molto alta), unitamente a una percezione elevata dell'utilità del feedback. Gli utenti hanno descritto l'interazione come più chiara e meno frustrante, evidenziando una maggiore sensazione di cooperazione e mostrando come l'adattamento sensibile al contesto trasformi l'HDI da semplice esecuzione di comandi isolati a processo collaborativo in continua evoluzione.

Parole chiave: Interazione Uomo-Drone (HDI), Interazione Uomo-Robot (HRI), Modelli Linguistici di Grandi Dimensioni (LLM), Interfacce Adattive, Sistemi Sensibili al Contesto.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **vii**

1 Introduction **1**

1.1 Limitations . . . 2
1.2 Contribution . . . 3
1.3 Thesis Outline . . . 5

2 State of the Art **7**

2.1 Human-Drone Interaction . . . 7
2.1.1 Human-Robot Interaction: Foundations and Evolution . . . 8
2.1.2 From HRI to HDI . . . 9
2.1.3 Domains and Applications . . . 10
2.2 Human-Drone Interfaces . . . 11
2.2.1 Available Interfaces . . . 12
2.2.2 Direct Interfaces . . . 13
2.2.3 Semi-Autonomous and High-Level Control . . . 13
2.3 Evolving Perspectives in Human-Drone Interaction . . . 15
2.3.1 Beyond Interaction: Mutual Shaping . . . 15
2.3.2 Temporal Evolution and Adaptation . . . 16
2.4 Large Language Models and Cognitive Architectures in HDI . . . 18
2.4.1 From Symbolic AI to LLM-Driven Reasoning . . . 18
2.4.2 LLMs in Human-Robot Interaction . . . 19
2.4.3 LLMs and VLA Systems for UAVs . . . 21
2.4.4 Online Semantic Planning . . . 21
2.5 Limitations and Motivation . . . 22

2.5.1 Identified Gaps in Current Approaches . . . 22
2.5.2 DronWO: A Writable HDI Baseline . . . 24
2.5.3 TypeFly: A Cognitive HDI Baseline . . . 25
2.6 Summary and Synthesis . . . 26
2.6.1 From Control to Cognitive Collaboration . . . 26
2.6.2 Positioning DronWO and TypeFly . . . 27

3 TypeFly 29
3.1 Overview . . . 29
3.2 System Architecture . . . 29
3.3 System Components . . . 30
3.3.1 MiniSpec . . . 31
3.3.2 User Task in Natural Language . . . 31
3.3.3 Context Representation . . . 32
3.3.4 Robot Capabilities . . . 32
3.3.5 Prompt Generator and MiniSpec Runtime . . . 34
3.3.6 Prompt Engineering . . . 35
3.3.7 Interaction During Execution . . . 36

4 Design of an Adaptive, Context-Aware Interface 39
4.1 Design Motivation . . . 40
4.1.1 Limitations of Existing Solutions . . . 40
4.1.2 Solution Overview . . . 42
4.2 Illustrative Example: *Find an Apple* . . . 45
4.3 Our Solution - DACS . . . 47
4.3.1 Conceptual Model . . . 47
4.3.2 Prompting Strategy . . . 48
4.3.3 High-Level Skill Creation . . . 49
4.3.4 Memory . . . 51
4.3.5 Visual Environment Awareness . . . 57
4.3.6 User Clarification in Ambiguous Tasks . . . 58
4.3.7 Flyzones . . . 59
4.3.8 Integrated Example . . . 61

5 Implementation 63
5.1 Baseline TypeFly Implementation . . . 64
5.2 Software Architecture . . . 65
5.2.1 Hardware Abstraction Layer . . . 66

5.2.2 Skill Layer . . . 66
5.2.3 Execution Layer . . . 67
5.2.4 Reasoning Layer . . . 67
5.2.5 LLM Instance Abstraction . . . 68
5.3 Environmental Internal Representation . . . 69
5.4 LLM Instances . . . 73
5.4.1 Instances Configuration . . . 73
5.4.2 Plan LLM Instance . . . 74
5.4.3 Query LLM Instance . . . 77
5.4.4 Short-Term Memory LLM Instance . . . 78
5.4.5 Save Task Feedback LLM Instance . . . 79
5.4.6 Update Universal Feedback LLM Instance . . . 82
5.4.7 Choose Direction LLM Instance . . . 82
5.4.8 Create Graph LLM Instance . . . 84
5.4.9 Create Flyzone LLM Instance . . . 86

6 Prototyping 89
6.1 Overview . . . 89
6.2 DJI Tello Drone . . . 89
6.3 Crazyflie and Lighthouse Positioning System . . . 91
6.4 Edge Server . . . 92

7 Evaluation 93
7.1 Goals and Research Questions . . . 93
7.2 Overview . . . 94
7.3 Participants . . . 95
7.3.1 Pre-Test Profiling . . . 95
7.4 Evaluation Protocol . . . 96
7.4.1 Physical Environment . . . 97
7.4.2 Interaction Modalities . . . 98
7.4.3 Interaction Scenarios . . . 98
7.5 Results . . . 102
7.5.1 System Self-Explainability . . . 102
7.5.2 Feedback Mechanisms . . . 106
7.5.3 Context Graph Memory . . . 108
7.5.4 Flyzones and Safety . . . 110
7.5.5 Directional Visual Reasoning . . . 113
7.5.6 Shortcuts . . . 115

7.5.7 User-perceived Latency 117
7.5.8 High-Level Skills 120
7.6 Summary 122

8 Conclusion **125**

Bibliography **127**

A Questionnaires **133**

List of Figures **139**

List of Tables **143**

1 | Introduction

Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, have evolved from expert-operated platforms into increasingly autonomous systems used across domains such as inspection, logistics, emergency response, and media production. Advances in sensing, onboard computation, and artificial intelligence have shifted research attention from low-level flight control toward the broader question of how humans interact with aerial robots. Human-Drone Interaction (HDI) emerges within the wider field of Human-Robot Interaction (HRI), which frames interaction not only as command execution but as a cognitive and social process shaped by perception, context, and mutual adaptation. Suchman's notion of *"from interactions to integrations"* [53] captures this transition, emphasizing collaboration as an emergent activity within shared contexts of action.

HDI extends these ideas to aerial systems, where interaction occurs through spatial movement, perception, and multimodal communication rather than physical manipulation. Early approaches focused on manual or gesture-based control, while later work introduced shared autonomy, enabling users to express higher-level objectives that drones execute autonomously. More recently, the integration of Large Language Models (LLMs) and Vision Language Action (VLA) architectures has enabled drones to interpret natural-language goals and adapt behavior dynamically.

The evolution of HDI can be conceptualized as the trajectory illustrated in Figure 1.1. Moving from manual control toward shared autonomy and language-driven reasoning, the field increasingly explores interaction paradigms that integrate adaptive behaviors with contextual understanding. Within this trajectory, *writable interaction* approaches extend the interaction space by allowing users to progressively shape the interface according to their needs, effectively *writing* new executable behaviors during use rather than relying on a fixed set of predefined commands. In contrast, LLM-driven approaches emphasize reasoning-based autonomy, where the system interprets natural-language goals and generates executable plans without requiring users to explicitly construct action sequences. These complementary directions highlight a broader transition within HDI: from control-centric interfaces toward cognitive collaboration between human and drone.

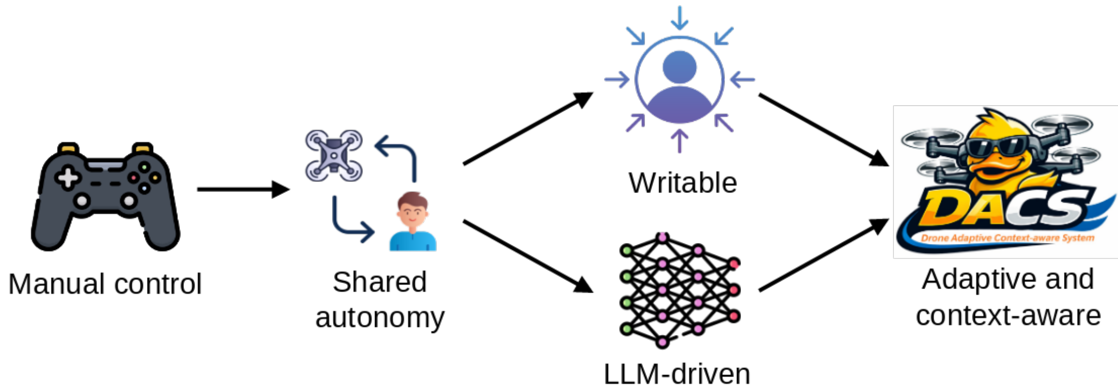


Figure 1.1: Evolution of Human-Drone Interaction paradigms.

Within this trajectory, the paradigms illustrated in Figure 1.1 highlight complementary directions in the design of HDI systems. The *Writable* direction emphasizes user authorship and interface extensibility, enabling users to construct new behaviors through interaction. Systems such as `DronWO` exemplify this approach by allowing users to define executable actions through gesture-based composition. In contrast, the *LLM-driven* direction focuses on reasoning-based autonomy, where users specify goals in natural language and the system generates executable plans automatically. Frameworks such as `TypeFly` represent this paradigm, prioritizing cognitive adaptation over explicit authoring.

Together, these systems highlight a broader transition within HDI: from control-centric interfaces toward cognitive collaboration. At the same time, they reveal a gap between expressive writability and adaptive reasoning.

1.1. Limitations

Building on the landscape outlined above, current HDI research still presents a key fragmentation between expressive interaction and cognitive autonomy. Writable interfaces enable users to actively shape drone behavior, whereas reasoning-driven systems prioritize goal-oriented communication and automated planning. However, these two perspectives rarely converge within a single framework.

As conceptually illustrated in Figure 1.1, these two directions represent complementary yet incomplete perspectives. `DronWO` demonstrates how users can *write* new executable actions by defining gesture-based sequences, turning interaction into a process of active authorship. While this enables personalization and flexible composition of behaviors, it also introduces cognitive overhead and lacks mechanisms for contextual reasoning or adaptive evolution. In contrast, `TypeFly` shifts the focus toward natural-language interaction,

allowing users to concentrate solely on describing their goals while the system translates intent into executable plans. Although this reduces interaction complexity, it limits user-driven extension and provides little support for long-term adaptation or context-aware learning.

The coexistence of these approaches reveals a broader limitation: current HDI systems either empower users to construct behaviors or autonomously infer them through reasoning, but rarely integrate both capabilities. As a result, interaction remains either highly expressive but rigid, or flexible but non-authorable.

1.2. Contribution

The contribution of this thesis is to bridge this divide through a context-aware HDI paradigm that combines writable interaction with reasoning-based autonomy. The proposed approach integrates natural-language communication, contextual grounding, and mechanisms for user-driven extension within a unified architecture. By merging the authorability of writable systems with the flexibility of LLM-driven planning, it aims to support drones that evolve together with users across tasks and over time.

To address the limitations discussed in the previous sections, we introduce `DACS (Drone Adaptive Context-aware System)`, an adaptive and context-aware framework designed to turn natural-language drone interaction into a continuous learning process. Instead of treating each command as an isolated event, the system integrates reasoning, memory, perception, and user feedback within a unified loop that evolves across time and tasks.

Figure 1.2 illustrates the conceptual model guiding the design of `DACS`. At the center lies a Large LLM that acts as the reasoning core of the drone. The LLM mediates the interaction between four elements: the *user*, who expresses goals through natural language; the *memory*, which accumulates experience and preferences; the *environment*, perceived through onboard sensing and visual information; and the *drone* itself, which executes actions and feeds outcomes back into the system. Rather than separating interaction, perception, and control into independent pipelines, this model emphasizes a continuous perception-action loop in which contextual knowledge and user intent jointly shape behavior.

To clarify how the proposed mechanisms operate together, consider a simple yet representative instruction: *"find an apple"*. In the baseline `TypeFly`, this request results in a predefined scanning routine that is repeated until the object is detected, regardless of past experience or environmental context. In contrast, `DACS` approaches the task as an evolving process shaped by interaction history, visual cues, and user input.

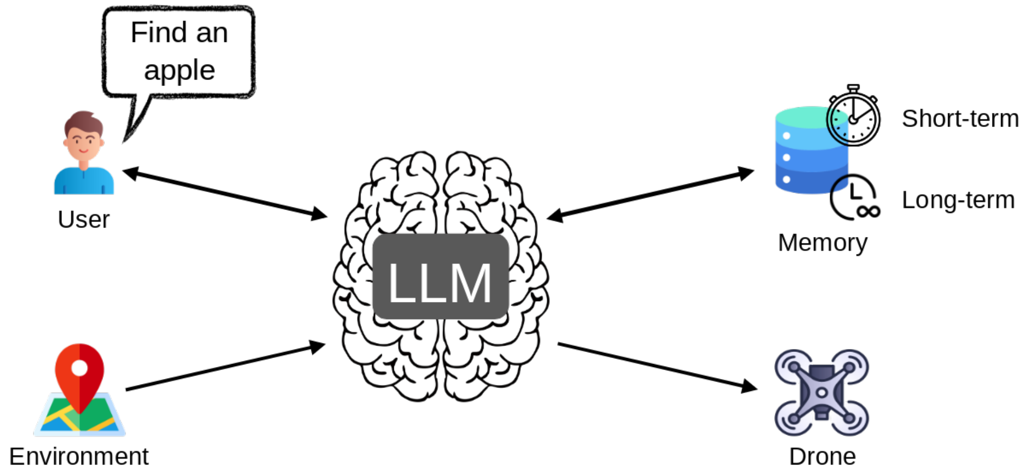


Figure 1.2: Conceptual model of `DACS`. The LLM acts as the central reasoning component, mediating interaction between the user, memory, environment, and drone.

Interaction becomes dialogical rather than one-directional through the *user clarification* mechanism. When a request is ambiguous or incomplete, the system can ask follow-up questions before generating a plan, ensuring that execution aligns with the user's intention. During execution, the `Short-Term Memory` maintains a dynamic trace of actions and observations, allowing the reasoning process to adapt as new information emerges instead of restarting from a fixed routine. Across multiple sessions, the `Long-Term Memory` accumulates knowledge derived from *user feedback*, reusable behaviors, and environmental exploration. For example, if apples were previously detected in a kitchen area, the drone can prioritize that region rather than performing a blind search.

Perception is also extended through `Visual Environment Awareness`. Beyond symbolic object detection, images captured by the onboard camera can be interpreted by the LLM to infer contextual cues even when the target object is not directly visible, such as recognizing that a fruit bowl or a kitchen surface suggests a likely search location. This integration of visual context transforms exploration from repetitive scanning into a reasoning-driven process guided by semantic relations between objects and environments.

As interaction continues, the system progressively encapsulates successful task executions into reusable `High-Level Skills`. Instead of regenerating the same plan from scratch, the drone can invoke previously validated behaviors, improving both efficiency and consistency. These skills evolve through *user feedback*, which allows the user to refine the drone's behavior over time; for instance, a request such as *"next time, tell me how many apples you see"* becomes part of the system's long-term knowledge and influences future planning decisions.

Finally, reasoning remains grounded in real-world constraints through the notion of *fly-zones*, which allow users to define safe operating regions directly in natural language. These spatial boundaries provide contextual structure for planning, ensuring that adaptive exploration remains predictable and aligned with safety requirements.

Together, these mechanisms transform the interaction paradigm from a static execution pipeline into a continuously evolving collaboration between human and drone. Rather than merely interpreting commands, `DACS` integrates memory, perception, and dialogue to progressively refine its behavior, enabling a more natural and context-aware form of human-drone interaction.

To validate the proposed approach, we developed a research prototype of `DACS` deployed on a real aerial platform. The system combines a DJI Tello quadrotor for actuation and visual sensing with a Bitcraze Crazyflie 2.1 equipped with a Lighthouse positioning deck to provide accurate indoor pose estimation, while perception, planning, memory management, and LLM reasoning are executed offboard on an edge server. Users interact with the drone through a web-based chat interface, issuing natural-language commands and receiving textual feedback, while additional visualizations expose internal elements such as flyzones and explored regions.

We evaluated the prototype through a comparative user study involving ten participants divided into two groups, each interacting with either the baseline `TypeFly` or the adaptive `DACS` system. The evaluation adopts an open-ended interaction protocol in which users explore tasks such as object search, repeated commands, and environment-aware navigation, allowing the study to assess usability, adaptation through memory, and user-driven customization mechanisms in realistic scenarios. Overall, the findings suggest that, in the baseline `TypeFly`, the drone often became stuck executing repetitive scanning or replanning routines when the target object was not immediately detected. In contrast, `DACS` leveraged short-term memory, visual environment awareness, and accumulated feedback to adapt its strategy over time, enabling the drone to move beyond fixed routines and continue exploration based on contextual clues. Participants reported that this behavior made the system feel more responsive and collaborative, providing initial evidence that integrating memory, dialogue, and context-aware reasoning can transform HDI from static command execution into an adaptive interaction process.

1.3. Thesis Outline

The following chapters develop the contributions of this thesis, moving from background and baseline systems toward the design and evaluation of `DACS`:

- **Chapter 2** introduces the theoretical and technical foundations underlying this work, covering HRI, HDI, writable interfaces, and recent advances in LLM-driven reasoning and multimodal perception. This chapter establishes the conceptual and technological context within which the proposed system is positioned.
- **Chapter 3** presents the baseline `TypeFly` system, describing its architecture, interaction model, and planning pipeline. It highlights the strengths of natural-language goal specification while exposing the limitations that motivate the adaptive approach developed in this thesis.
- **Chapter 4** introduces the design of `DACS`, detailing the conceptual model and the mechanisms that enable adaptive and context-aware interaction, including Short-Term and Long-Term Memory, Visual Environment Awareness, High-Level Skill creation, user clarification strategies, and flyzone-based spatial reasoning.
- **Chapter 5** describes the software implementation of `DACS`, detailing the modular architecture, data flow, memory management mechanisms, perception and reasoning pipelines.
- **Chapter 6** presents the research prototype and experimental platform used to validate the system. It details the hardware configuration, aerial platform selection, and the distributed architecture adopted to separate real-time flight control from computationally intensive processing.
- **Chapter 7** presents the user study conducted to evaluate the system, detailing the experimental design, tasks, and analysis of both quantitative and qualitative findings to assess usability and adaptive behavior.
- **Chapter 8** concludes the thesis by summarizing the main contributions, discussing limitations, and outlining directions for future research in adaptive HDI.

2 | State of the Art

This chapter presents an overview of the current state of the art in **Human-Drone Interaction (HDI)**, situating the research developed in this thesis within the broader evolution of the field. It traces the progression from early investigations in **Human-Robot Interaction (HRI)**, where communication between humans and machines was primarily defined by control accuracy and interface design, to contemporary paradigms that integrate perception, reasoning, and natural language through **Large Language Models (LLMs)** and **Vision Language Action (VLA)** architectures.

The chapter is organized into six sections that reflect the conceptual and technological development of the field. Section 2.1 outlines the transition from HRI to HDI, discussing how aerial systems became interactive partners in shared environments and summarizing the principal domains and applications where HDI has been applied. Section 2.2 examines how humans communicate with drones through various modalities, including gesture, voice, and multimodal interfaces, and how these have evolved from direct control schemes toward semi-autonomous configurations. Section 2.3 expands this discussion by introducing embodied and relational perspectives, such as *soma design*, *mutual shaping*, and *temporal adaptation*, which emphasize the co-evolution between humans and drones. Section 2.4 reviews the integration of cognitive reasoning into HDI through LLM-based and multimodal architectures, highlighting recent frameworks that combine perception, planning, and language understanding. Section 2.5 then identifies the main limitations of existing approaches and introduces two systems that serve as experimental baselines: `DronWO` [13], which explores embodied and writable interaction, and `TypeFly` [19], which extends those principles toward reflective autonomy driven by reasoning provided through an LLM. Finally, Section 2.6 concludes with a synthesis of these developments, outlining how the field's evolution motivates the design of the solution proposed in this thesis.

2.1. Human-Drone Interaction

This section outlines the conceptual foundations and historical evolution of HDI, situating it within the broader field of HRI. It first reviews the origins of HRI and its progressive

shift from command-based control to social and collaborative paradigms. It then discusses how HDI emerged as a specialized subfield addressing the spatial and perceptual particularities of aerial systems, and concludes with an overview of the principal domains and applications that have shaped current research directions.

2.1.1. Human-Robot Interaction: Foundations and Evolution

HRI is a multidisciplinary research domain concerned with how humans and robots communicate, collaborate, and share environments. Drawing on computer science, cognitive psychology, artificial intelligence, and design research, it seeks to create systems that are not only functional but also intelligible, adaptive, and trustworthy partners for humans. Interaction is therefore understood as a reciprocal process in which information, behavior, and intention circulate between human and robot rather than as a simple sequence of commands and responses [31].

Early HRI studies, from the 1960s to the 1980s, focused on teleoperation and supervisory control, particularly in industrial contexts. Robots were treated as precise yet passive tools operated through panels, joysticks, or textual interfaces, and interaction was evaluated mainly in terms of accuracy and repeatability. While these systems were effective in controlled settings, they imposed high cognitive demands on operators and offered limited adaptability to unstructured environments. The introduction of supervisory control shifted emphasis from manual actuation to higher-level task definition, establishing the conceptual basis for mixed-initiative systems that would later define modern HRI [49].

From the 1990s onward, advances in artificial intelligence, sensing, and computer vision extended robotics into social and service domains. Researchers began to investigate how robots could communicate intent, express emotion, and build trust through legible motion and transparent behavior. Interaction became recognized as a cognitive and social process influenced by perception, affect, and context rather than by technical precision alone. Suchman's seminal notion of *"from interactions to integrations"* [53] captured this conceptual shift, framing collaboration as an emergent and situated activity within shared contexts of action. The focus of HRI thus moved from command accuracy to embodied coordination and mutual adaptation.

Within this broader transformation, aerial robots introduced a new class of challenges. Operating in three-dimensional space, drones must maintain stability, avoid collisions, and interact with humans without physical contact. These characteristics required a reformulation of central HRI principles, particularly transparency, trust, and safety, in contexts where communication occurs through motion and spatial behavior rather than

physical manipulation. This progression from control to cooperation, and from precision to understanding, provided the conceptual foundation for HDI as a distinct field of research.

2.1.2. From HRI to HDI

HDI emerged as a focused research area addressing the distinctive dynamics of aerial systems. Unlike terrestrial robots, which often rely on physical contact, grasping, or fixed-base manipulation, drones interact with humans primarily through visual, auditory, and spatial cues. Because drones move and communicate without touch, their behavior must be inferred from patterns of motion, sound, and proximity. This mode of interaction makes them simultaneously more expressive and more ambiguous: flight trajectories, hovering, and sound can convey intention and emotion vividly, but they also leave greater room for interpretation and misreading.

HDI retains several guiding principles from HRI, including *legibility*, *transparency*, and *shared autonomy*, yet reinterprets them through the perceptual and spatial logic of flight. *Legibility* [23] refers to how clearly a robot's movements reveal its intent to a human observer, enabling prediction of future actions; in aerial systems, this is often achieved through smooth, anticipatory trajectories that signal cooperation rather than threat. *Transparency* [44, 59] concerns the ability of the system to communicate its internal state, goals, or reasoning processes, for instance by signaling whether it is following a user, avoiding obstacles, or executing an autonomous task, thereby supporting user understanding and trust. *Shared autonomy* [22, 49] describes a control paradigm in which decision-making is distributed between human and robot, with the human specifying high-level goals and the autonomous system managing low-level execution, navigation, and safety constraints. Together, these principles support mutual predictability and cooperation, transforming drone flight into a form of non-verbal dialog.

Empirical studies such as `Drone & Me` [15] and `Drone Near Me` [1] show that people naturally attribute social and emotional meaning to drone behavior. Participants interpret distance, speed, and orientation as signs of curiosity, hesitation, or caution, establishing the foundations of *proxemic interaction* [60], in which spatial relationships act as communicative cues. Subsequent research extends this perspective by addressing ethical concerns, aesthetic qualities, and the environmental implications of human–drone cohabitation in aerial environments. According to Cauchard et al. [16], the key challenge is to design drones that are not only autonomous but also *socially intelligible*, capable of operating safely while maintaining mutual awareness with nearby humans.

Technological and regulatory developments have also shaped HDI. Advances in lightweight processors, onboard sensing, and adaptive control have enabled stable and responsive flight in both indoor and outdoor environments. At the same time, aviation regulations such as Beyond Visual Line of Sight (BVLOS) constraints [20] have introduced new requirements for safety and situational awareness. Historical reviews of unmanned aerial systems [34] show that technological innovation and regulatory responsibility have evolved together, continuously influencing how humans interact with aerial robots.

In summary, HDI extends the cognitive and social foundations of HRI into the aerial domain, where interaction depends on interpreting motion, space, and perception rather than physical contact. It reframes collaboration as a process of shared spatial negotiation, in which humans and autonomous agents coexist and coordinate their actions within the same three-dimensional environment.

2.1.3. Domains and Applications

HDI enabled its deployment across a wide range of domains, each characterized by distinct human-system relationships. A scoping review by Herdel et al. [32] identifies six primary areas: *emergency response*, *logistics and delivery*, *inspection and maintenance*, *entertainment and media*, *education and research*, and *personal assistance*. In each case, drones serve as intermediaries between human perception and the environment, augmenting human capability while introducing new interaction challenges.

In emergency and rescue operations, drones offer rapid situational awareness in hazardous or inaccessible environments. Research highlights the need for trust, mutual understanding, and reliable handover between manual and autonomous control [5, 37, 58]. Industrial inspection and construction contexts benefit similarly from multimodal collaboration: voice- or gesture-assisted control improves coordination and safety by reducing operator workload [62].

In these safety-critical domains, where trust and reliable handover between manual and autonomous control are essential, failures in HDI are not limited to physical crashes such as collisions or loss of stability. Breakdowns may also occur at the interactional level, when a drone behaves in ways that diverge from user expectations. Unexpected motion, unclear autonomy transitions, or delayed responses can lead operators to misinterpret intent, hesitate, or intervene inappropriately, sometimes exacerbating already hazardous situations [56]. Even in the absence of a physical incident, such mismatches between drone behavior and the user's mental model are often perceived as errors, undermining trust and situational awareness and increasing the likelihood of unsafe responses [15, 40].

Creative and educational uses of HDI explore expressivity rather than efficiency. In interactive art and classroom settings, drones act as dynamic artifacts that visualize concepts of rhythm, balance, or cooperation. Here, interaction breakdowns take a different form: abrupt or unintelligible behavior can disrupt engagement or diminish the user's sense of *agency*, that is, the feeling of being able to meaningfully influence the drone's behavior and outcomes through one's actions [27]. Such applications emphasize accessibility and play, showing how humans quickly form emotional and aesthetic attachments to aerial agents, reinforcing the importance of designing socially legible behavior.

Despite this diversity, most HDI systems remain domain-specific and difficult to generalize. Approaches optimized for emergency response may not transfer to domestic or artistic scenarios, where spatial, temporal, and social expectations differ. Comprehensive surveys [32, 56] and the roadmap by Cauchard et al. [16] therefore call for integrated frameworks that unify sensing, reasoning, and human factors. Establishing shared evaluation protocols and benchmarks that account for both physical safety and interactional robustness remains a key prerequisite for progress.

The next generation of HDI research thus converges on two main objectives: (i) developing architectures that integrate perception, language, and reasoning across modalities, and (ii) creating empirical methods that assess adaptability and user alignment over time. These goals motivate the investigation of reasoning-based autonomy frameworks such as `TypeFly`, presented in Chapter 3, which serves here as a reference baseline for reflective, cognitively grounded HDI.

2.2. Human-Drone Interfaces

Human–drone interfaces define how people communicate with aerial systems and how these exchanges shape both control and understanding. The interface determines not only performance but also how meaning, trust, and feedback circulate within the collaborative loop. This section examines interfaces along three main dimensions: the *modality* of communication (gesture, voice, or gaze), the *level of abstraction* (from low-level control to high-level task specification), and the *distribution of agency* [27] between human and system. We first review the available sensing modalities and configurations, then discuss direct control mechanisms, and finally consider semi-autonomous and adaptive approaches that integrate reasoning and supervision.

2.2.1. Available Interfaces

Human-drone interfaces vary according to how sensing and computation are distributed between the user and the platform. *Onboard interfaces* rely on sensors integrated into the drone, such as RGB cameras, microphones, inertial units, or depth sensors, to infer human intent directly from the surrounding environment. In contrast, *external interfaces* depend on fixed infrastructure such as motion-capture arrays, wearable sensors, or vision-based tracking systems. Although external setups can achieve higher positional accuracy, they reduce portability and *ecological validity* [11], defined as the extent to which an interaction system supports behavior that is representative of real-world use conditions. As a result, such interfaces often perform well in controlled laboratory settings but may not generalize effectively to everyday environments, where lighting, space constraints, mobility, and user behavior are less predictable.

A second distinction concerns *intrusiveness*. Intrusive systems require users to wear or hold devices, whereas non-intrusive systems infer gestures or posture through cameras or radar sensing. Each option involves trade-offs: non-intrusive setups preserve freedom of movement but are sensitive to occlusion and lighting, while wearable systems ensure robustness but can restrict expression or cause fatigue [25].

The development of *Natural User Interfaces (NUIs)* has reshaped HDI by reducing the cognitive translation between intention and action. NUIs use instinctive human behaviors such as gesture, gaze, posture, or speech to elicit drone responses. Experiments on near-body interaction show that drones evoke spatial behaviors similar to interpersonal proxemics, as people adjust distance and orientation according to the drone's perceived responsiveness or personality [1, 15]. Designers have therefore begun to encode approach, retreat, and hovering patterns as communicative gestures, transforming motion itself into a language of interaction.

These embodied paradigms redefine the drone's role by shifting it from a passive tool to what the literature describes as a perceived *living entity*. Rather than functioning solely as an instrument, the drone is experienced as an active interlocutor whose influence, expressed through movement, behavior, and responsiveness, can resemble that of a living entity within interaction contexts [61]. The perception of *animacy*, defined as the quality that makes artificial agents appear lifelike rather than inert, further explains why humans often treat robots and drones as social entities rather than mere machines [14, 21]. This perception of the drone as a living entity raises questions of safety and interpretability: how can drones make their intentions legible, and how can humans accurately anticipate their actions?

2.2.2. Direct Interfaces

Direct interfaces recreate the immediacy of manual control through natural modalities. Human gestures or body movements are mapped directly to drone actions, preserving a sense of embodied connection. Examples include accelerometer-based wearables [46], multi-drone coordination through arm gestures [52], and posture-based navigation systems [30]. These studies confirm that physical engagement reduces the cognitive gap between intention and actuation, although ergonomics and recognition reliability remain limiting factors.

The advantage of direct control lies in its intuitiveness: users can operate drones with little or no prior training. However, prolonged use can lead to fatigue, and precision decreases with distance or limited visibility. Because motion inputs are translated into actions without semantic reasoning, direct interfaces represent the lowest level of autonomy, where the drone behaves as an extension of the user's body rather than as an independent partner.

To enhance robustness and expressiveness, recent work has combined gesture, voice, and gaze into multimodal control schemes. For example, Zhu et al. [62] shows that voice confirmation can reduce recognition errors and improve safety during collaborative construction tasks. Such multimodal systems mark an evolution from purely embodied control toward interfaces that incorporate perception and contextual understanding. By integrating sensory feedback and limited environmental awareness, these systems begin to bridge the gap between direct manipulation and adaptive, semi-autonomous interaction.

2.2.3. Semi-Autonomous and High-Level Control

As autonomy increases, the human role shifts from direct manipulation to supervision and decision-making at higher levels of abstraction. *Semi-autonomous interfaces* mediate this balance by distributing control: humans define goals or constraints, while drones handle low-level motion, obstacle avoidance, and safety. This arrangement reduces operator workload and enhances efficiency without removing human oversight.

Early research on shared control illustrates this model. Cacace et al. [12] develops a multimodal search and rescue system that integrates voice, gesture, and semi-autonomous behaviors. Operators could coordinate multiple drones using intuitive commands while a supervisory architecture validates task feasibility and monitors safety signals before execution, providing software-level safeguards rather than direct low-level control. These safeguards ensure that even ambiguous or incomplete instructions are executed within safe

limits. Similarly, Medeiros et al. [45] demonstrates pointing-based interaction, where gaze or arm direction designates a target region of interest. In both cases, symbolic mapping of user intention replace continuous manual control, improving clarity and reducing cognitive load.

High-level interfaces extend this principle by allowing the degree of autonomy to adapt dynamically. In predictable conditions, the drone can act independently, whereas in uncertain contexts, control reverts to the human operator. This adaptability aligns with HRI theories of *mixed-initiative interaction*, in which initiative shifts according to confidence, trust, or environmental risk [7, 31]. For aerial platforms, such flexibility is crucial because visibility, stability, and safety vary continuously with environmental conditions.

At the same time, these dynamic autonomy shifts introduce a characteristic source of failure in semi-autonomous HDI. When transitions of initiative are unexpected, insufficiently communicated, or poorly aligned with the user's mental model, drone behavior may diverge from what the user anticipates, even if low-level safety constraints are respected. In such cases, crashes manifest not only as physical incidents but also as interaction breakdowns arising from autonomy negotiation, where users may hesitate, overcorrect, or intervene inappropriately in response to unanticipated behavior. This highlights that effective semi-autonomous control depends not only on collision avoidance or constraint enforcement, but also on making autonomy boundaries, system intent, and decision processes legible throughout interaction.

Two notable systems illustrate complementary directions within this spectrum. The `DronWO` system, described in Section 2.5.2, explores the concept of a *writable interface*, where users can compose new actions by chaining gestures into user-defined sequences. These sequences can be mapped to custom behaviors, such as specific trajectories, created during interaction. Rather than embedding autonomy, `DronWO` emphasizes user authorship and creative control, demonstrating how interaction itself can become a medium for expression.

By contrast, the `TypeFly` system, described in Chapter 3, focuses on cognitive understanding through reasoning-based autonomy, using an LLM to interpret natural-language goals and generate structured action plans. Together, these systems delineate two complementary paths for HDI: one centered on expressive embodiment and user authorship, and the other on cognitive understanding and language-driven reasoning.

The transition from direct to semi-autonomous interaction also introduces new feedback challenges. Users must understand the drone's goals, progress, and confidence without being overwhelmed by information. Recent studies explore multimodal cues such as sound,

haptic vibration, or augmented-reality overlays to communicate system status and autonomy boundaries. Maintaining a coherent mental model of the drone's behavior remains essential so that users can anticipate when and why the drone takes initiative.

2.3. Evolving Perspectives in Human-Drone Interaction

Research in HDI has progressively expanded beyond issues of control precision and responsiveness toward more relational and experiential questions. As drones acquire perceptual, reasoning, and adaptive capabilities, interaction can no longer be described as a one-way exchange of commands from the user to the drone. Instead, it unfolds as a dialog of perception and movement between embodied agents, one human and one robotic. This section explores how HDI has evolved through the lenses of embodiment, somaesthetic design, and temporal co-adaptation, outlining how these perspectives inform the transition toward reflective and cognitively enriched interaction.

2.3.1. Beyond Interaction: Mutual Shaping

Traditional approaches to HRI and HDI viewed interaction as a sequential process in which the human issued commands, the robot executed them, and feedback closed the loop. Over time, researchers and designers have challenged this asymmetry, drawing on the philosophy of technology, embodied cognition, and design research to propose the concept of *mutual shaping*. In this view, humans and machines continuously influence each another, co-evolving their perceptions, expectations, and behaviors through ongoing collaboration.

In HDI, mutual shaping foregrounds the affective and kinesthetic dimensions of interaction. A drone's flight path, distance, and dynamics do not simply respond to human input but also elicit bodily and emotional responses. A hovering drone at eye level may appear curious or companion-like, while abrupt lateral motion can feel defensive or intrusive. Subtle variations in timing and motion define what has been termed the drone's *expressive bandwidth*, meaning its capacity to convey emotion and intent through movement. This expressive capacity transforms the drone from a mechanical tool into a social artifact that inhabits a shared perceptual space with humans [7, 33].

Soma design provides a conceptual foundation for this approach. Introduced by Hook [33], soma design treats interaction as a process of cultivating bodily awareness and aesthetic sensitivity rather than optimizing performance. It invites designers to focus on the felt

experience of movement, texture, and rhythm as users engage with technology. Applied to drones, this perspective encourages exploration of how flight can resonate with the user's own gestures, how acceleration, hovering, or sway can produce sensations of harmony, tension, or empathy. The aim is not to maximize control accuracy but to create an embodied dialog in which human and drone movements co-shape one another's sense of balance and presence.

The notion of *mechanical sympathy* extends soma design to aerial systems [38]. It emphasizes physical co-adaptation, the ability of human and drone to move together through mutual anticipation, similar to dancers synchronizing to a shared rhythm. Here, effective interaction depends not on explicit commands but on sensitivity to timing, inertia, and responsiveness. This perspective reframes flight as a choreography of understanding, where human skill and machine dynamics merge into coordinated flow.

Related research introduces the concept of the drone's *umwelt*, or perceptual world, as a metaphor for design [39]. Each agent, human or drone, perceives the world through its own sensory filters and operational logic. Designing for mutual shaping therefore means creating interfaces that respect and bridge these perceptual asymmetries. By aligning human intention with the drone's perceptual limits and capabilities, interaction becomes a negotiation between two distinct yet connected modes of sensing.

Extending these ideas into computation, Sondoqah et al. [51] describes *programming in perception-action loops*, a paradigm in which both human and drone continuously adjust their behavior in response to one another. Each becomes, in effect, a programmer of the other's behavior through reciprocal feedback. This blurs the boundary between user and system, establishing a framework for co-adaptation and shared agency that underpins the reflective architectures discussed later in this chapter.

2.3.2. Temporal Evolution and Adaptation

The ideas of bodily coordination and mutual shaping discussed in the previous subsection naturally extend into the temporal dimension of HDI. Interaction is not confined to a single moment but unfolds over time, as both human and drone gradually adjust to one another's patterns. Earlier HDI experiments often treated interaction as isolated episodes: a gesture triggered a response, and the exchange ended. With advances in learning, mapping, and perception, interaction is now viewed as an evolving relationship that develops across multiple encounters. Each session contributes to a shared history in which both partners accumulate expectations, routines, and behavioral refinements.

Understanding HDI as a temporally extended process means recognizing that adaptation

occurs reciprocally. Humans refine their gestures, phrasing, or positioning within the drone's perceptual field, while the drone modifies its internal models of human intent and environmental context. This dynamic mirrors the notion of *co-adaptation* established in HRI, where effective collaboration emerges from ongoing adjustment rather than fixed protocols. Empirical studies show that repeated interaction enhances predictability and trust, particularly when the system's feedback is clear and its behavior remains consistent over time.

Current approaches to adaptive HDI demonstrate this tendency toward temporal learning but remain mostly confined to the behavioral level. Many systems can fine-tune control parameters or respond to short-term cues yet fail to preserve knowledge across interactions. They adapt within a single session but do not develop a sense of continuity or experience. Consequently, interaction remains reactive rather than developmental, lacking the ability to accumulate understanding over longer timescales.

Addressing this limitation requires moving beyond instantaneous adaptation toward what can be described as *temporal co-adaptation*, a process in which human and drone mutually shape one another's behavior through sustained engagement [38, 51]. In this view, adaptation is not merely an optimization of performance within isolated episodes but the gradual emergence of shared expectations, interaction patterns, and embodied coordination across encounters, as users learn to anticipate and influence the drone's actions while the drone's behavior implicitly informs users' strategies. Empirical work on embodied human–drone interaction shows that participants' bodily movements and perceptual strategies evolve as they interact with drones, creating an iterative sense-making loop in which humans adjust their actions in light of the drone's responses and vice versa [51]. This type of mutual shaping highlights time as a critical factor in HDI, linking bodily attunement with cognitive anticipation and suggesting that long-term interaction cannot be understood as a series of independent episodes but as a co-evolutionary process of reciprocal adaptation.

As HDI research evolves from embodied synchrony toward temporally sustained cooperation, the central challenge becomes integration, combining perception, reasoning, and temporal awareness within a single framework. LLM architectures, discussed in Section 2.4, provide the foundations for this direction by enabling drones to interpret natural language, contextualize prior interactions, and reason about change. Developing systems capable of such temporal awareness represents an open frontier for HDI and provides the conceptual motivation for the architectures introduced later in this thesis.

2.4. Large Language Models and Cognitive Architectures in HDI

Recent advances in artificial intelligence have transformed the foundations of robotic autonomy. Where earlier drones relied on pre-programmed routines or teleoperated control, contemporary systems increasingly demonstrate the ability to interpret, reason, and learn from multimodal information. LLMs and related cognitive architectures bridge structured reasoning and embodied perception, enabling systems that integrate language, vision, and action within a single decision loop.

To understand this transition, it is useful to recall the era of *symbolic artificial intelligence* [26]. Symbolic AI refers to computational approaches that represent knowledge explicitly through symbols, logic rules, and ontologies. These systems reasoned by manipulating symbolic representations of the world, allowing clear and explainable decision-making but offering limited adaptability to noise or uncertainty. They provided strong analytical reasoning but lacked perceptual grounding, as they could not directly interpret sensory data such as images or movement. The subsequent rise of machine learning and neural networks replaced hand-crafted reasoning with data-driven pattern recognition, greatly improving perception and control yet often sacrificing transparency.

LLMs [17] represent a new synthesis of these traditions. Built on transformer architectures, they combine the generalization ability of neural models with the structural reasoning once associated with symbolic AI. When integrated with robotic perception and control modules, they act as cognitive cores that can interpret natural language, infer intent, and generate executable actions. This section traces that evolution, from early symbolic reasoning to multimodal VLA systems, and discusses how these advances provide the conceptual basis for context-aware and adaptive HDI.

2.4.1. From Symbolic AI to LLM-Driven Reasoning

For several decades, autonomy in robotics was shaped by symbolic artificial intelligence. Symbolic systems represented knowledge through logic rules, symbolic operators, and explicit planning graphs, enabling deterministic and interpretable decision-making. Although these planners are explainable, they lack perceptual grounding and often fail in dynamic or uncertain environments. In contrast, the rise of neural networks introduces data-driven models capable of learning perception and control directly from sensor data. These models excel at recognizing patterns but offer limited insight into their internal decision processes, creating a divide between *seeing* and *understanding*.

Transformer architectures, introduced by Vaswani et al. [57], established a new paradigm for sequence modeling grounded in attention mechanisms. Rather than relying on recurrence or convolution, transformers capture dependencies through *self-attention*, which evaluates how each element in a sequence relates to all others. Self-attention assigns a learned weight to these relationships, enabling the model to focus selectively on relevant contextual information. According to the original definition, the transformer "dispenses with recurrence and convolutions entirely and relies solely on attention mechanisms to draw global dependencies between input and output." This design allows transformers to capture long-range context, model complex dependencies, and process linguistic or visual information in parallel, improving scalability and generalization.

When integrated into robotic pipelines, transformers function as cognitive cores that interpret natural-language instructions, reason about environmental context, and synthesize structured action plans. This capability establishes what can be described as *reasoning-based autonomy*: instead of executing predefined command patterns, the system derives its behavior through natural-language inference and contextual interpretation. In HDI, this shift enables users to express goals in everyday language, while the drone decomposes those goals into grounded sequences of actions. By verbalizing its intermediate reasoning, the system increases transparency and supports trust, forming the basis for more interpretable and adaptive aerial autonomy.

2.4.2. LLMs in Human-Robot Interaction

Integrating language models into robotics has produced a new class of systems that connect linguistic understanding with physical execution. One early milestone is `SayCan` [3], which couples an LLM with an affordance model that ranks possible robot skills according to semantic relevance and likelihood of success. Given a natural-language goal, the system selects actions that satisfy intent while remaining physically feasible, demonstrating that language reasoning can guide embodiment safely and predictably.

Building on this idea, `Code-as-Policies (CaP)` [41] treats the LLM as a program generator rather than a direct controller. Given a natural-language instruction, the model outputs short, human-readable code fragments composed of reusable *primitives*. For example, the instruction "pick up the red block and place it on the shelf" can be translated into:

```
move_to("red_block")
grasp("red_block")
move_to("shelf")
```

```
release("red_block")
```

Each primitive corresponds to a verified low-level controller, ensuring that the LLM sequences trusted actions rather than manipulating actuators directly. This mechanism turns the generated code into both an *interpretable policy* and a *reasoning trace* that reveals the system's logic. Because the model can recombine existing primitives in new ways, it exhibits *compositional generalization*, which is the ability to handle novel tasks by assembling known behaviors. `CaP` therefore unites the flexibility of neural policies with the modularity of symbolic reasoning, offering a transparent basis for language-driven autonomy.

A further step is `PaLM-E` [24], which processes images and text within the same transformer architecture. By interpreting visual and linguistic cues together, it performs grounded reasoning and can understand spatial relations such as "the cup to the left of the plate" or "move the red block to the table." This integration of perception and language lays the groundwork for more general multimodal control frameworks.

Subsequent large-scale efforts, including `RT-1` and `RT-2` [10, 63], extend these principles into what are now referred to as VLA models. A VLA model links three core components: visual perception that interprets the environment, language understanding that captures user intent, and action generation that produces appropriate motor behavior. Because these elements are trained jointly on large demonstration datasets, VLA systems learn direct mappings from observations and textual goals to motor policies, enabling zero-shot generalization across robotic embodiments.

`ProgPrompt` [50] further advances this direction by producing program-like plans composed of subgoals and conditions, which provide interpretable and auditable reasoning structures. More recently, DeepMind's `Gemini Robotics` [55] has introduced an embodied foundation model that unifies perception, reasoning, and control within a single multimodal network capable of cross-domain adaptation without task-specific retraining.

Together, these milestones chart the transformation of language models from passive text generators into active reasoning engines that orchestrate perception and control. They enhance transparency through verbalized rationales and flexibility through generalization, although challenges remain in grounding symbolic predictions in physical reality, ensuring reliability under uncertainty, and embedding ethical safeguards in embodied systems.

2.4.3. LLMs and VLA Systems for UAVs

The application of these cognitive architectures to aerial robots is recent but progressing quickly. Early experiments such as `From Words to Flight` [54] demonstrated how ChatGPT could interface with PX4 and Gazebo simulators through a middleware that validates and translates natural-language commands into flight instructions. The prototype achieves more than 90 percent accuracy on navigation and manipulation benchmarks, establishing an initial proof of concept for language-based UAV control and showing that natural-language reasoning can be operationalized in aerial contexts.

Javaid et al. [35] identifies three major research directions. The first concerns integrating LLM reasoning directly with flight controllers so that natural-language goals can be decomposed into safe and feasible motor actions. The second focuses on building multimodal datasets that combine visual observations with textual descriptions, enabling VLA models to ground language in the visual domain. The third explores planners that update their decisions online as environmental conditions change. These three directions share a common motivation: enabling drones that understand not only *what* action to execute but also *why* the action is appropriate in a particular spatial or semantic context.

More advanced prototypes extend these ideas beyond instruction following. For example, `CognitiveOS` [42] proposes a modular operating system that incorporates natural-language reasoning, structured memory, and ethical filtering through a method known as retrieval-augmented generation. In this method, the system retrieves relevant information from a curated memory or knowledge base and feeds it to the language model before generating an output. This ensures that the model reasons with accurate and context-specific information rather than relying solely on its internal training data.

`CognitiveDrone` [43] adapts this framework to aerial platforms and introduces the CognitiveDroneBench dataset for evaluating perception and language reasoning in UAV settings. Taken together, these developments portray drones as emerging cognitive agents that can ground visual input in semantic understanding and adjust their behavior dynamically in response to environmental change.

2.4.4. Online Semantic Planning

Many LLM-robot systems still reason in discrete cycles. They generate a plan, execute it, and only afterwards update their understanding of the environment. Real-world settings, however, change continuously and require constant adaptation. `SPINE` [48] addresses this limitation by combining LLM reasoning with advanced perception models such as

GroundingDINO and LLaVA.

GroundingDINO performs open-vocabulary object detection, identifying and localizing items in a scene through natural-language queries, for example "find the red cup" or "locate the person near the window," even when those objects were not part of its training data. LLaVA complements this functionality by interpreting visual scenes through conversational question answering.

Together, these perception models enable SPINE to construct and maintain an internal *semantic map* that links the spatial geometry of the environment with symbolic meaning. This map is updated continuously as the system observes new objects or changes in the scene, allowing the planner to revise its decisions in real time. Rather than executing a fixed plan, SPINE reasons over an evolving internal representation that reflects what the drone has seen and how the environment is changing.

By integrating high-level linguistic reasoning with real-time perceptual updates, SPINE connects symbolic understanding with embodied motion and achieves significantly higher efficiency than static planners. This approach marks an important step for HDI, where continuous perception, natural-language reasoning, and adaptive control must operate within a single coherent feedback loop.

2.5. Limitations and Motivation

The preceding sections have traced the evolution of HDI from manual and embodied control to reasoning-based autonomy. Although recent progress has significantly expanded the scope of HDI, current systems still face technical and conceptual limitations that restrict their adaptability, generality, and deployment in real-world contexts. This section consolidates those limitations and introduces two research platforms that serve as experimental and conceptual baselines for the work developed in this thesis: DronWO, which explores embodied and writable interaction, and TypeFly, which extends those principles toward reflective, reasoning-based autonomy. The systems are described here to frame the motivations that guide the design of the proposed approach, whose detailed architecture and evaluation are presented in the following chapters.

2.5.1. Identified Gaps in Current Approaches

The rapid diffusion of multimodal and language-based techniques has opened promising directions for HDI, yet several persistent challenges remain. A review of recent research highlights four recurring issues that limit scalability and robustness across domains.

The first challenge involves *adaptability*. Many aerial systems still behave as stateless agents, adapting their behavior only within the scope of a single task or interaction episode. Even advanced cognitive frameworks such as `CognitiveDrone` [43] include only within-task memory, storing information about the current execution to support immediate replanning but resetting once the task concludes. `SPINE` [48] represents a partial exception because it maintains a semantic map that persists throughout exploration, effectively creating a form of long-term environmental memory. However, this memory is largely limited to spatial and object-level information and does not extend to user-specific behavior, temporal patterns across tasks, or semantic interpretations of past interactions. Without mechanisms for preserving and updating experience across sessions, drones cannot accumulate behavioral continuity or progressively refine how they collaborate with individual users.

The second limitation concerns *the lack of unified cognitive architectures*. Most existing HDI systems address only a specific component of the interaction pipeline. Some focus primarily on user experience through intuitive interfaces such as gesture control or voice commands. Others prioritize robust online planning, spatial reasoning, or natural-language interpretation. Other approaches explore novel concepts such as writable interfaces that allow users to define behaviors on the fly. While each of these approaches advances an individual aspect of HDI, they rarely integrate perception, interpretation, decision-making, and control into a single coherent framework.

A third limitation concerns *environmental awareness*. Although perception models have advanced considerably, the connection between raw sensory data and symbolic interpretation remains fragile. In partially known or dynamic environments, drones must infer spatial relations among objects, regions, and actions, and update these relations continuously as conditions change. Frameworks such as `SPINE` demonstrate progress through open-vocabulary perception and dynamic semantic mapping, yet they still face challenges in maintaining globally consistent representations or using these representations for deeper semantic reasoning. A continuously evolving world model that supports both geometric awareness and semantic interpretation is essential for robust decision-making.

Finally, the issue of *safety and regulation* continues to impose practical constraints. Real-world HDI must comply with aviation standards and operate within strict safety envelopes. Beyond Visual Line of Sight (BVLOS) operations require accurate localization, reliable risk assessment, and well-defined spatial constraints [20], yet many experimental systems are tested in simulation or controlled indoor environments where these requirements are relaxed. Bridging the gap between controlled prototypes and deployable platforms requires architectures that incorporate safety logic, spatial boundaries, and

regulatory constraints directly into their reasoning process.

Taken together, these limitations highlight a broader fragmentation within current HDI systems. Most approaches achieve strong performance in isolated dimensions such as interaction, planning, or safety, but few integrate these capabilities coherently. Addressing this fragmentation requires cognitive architectures that unify embodied interaction, reasoning, memory, and environmental understanding. In this thesis, `DronWO` and `TypeFly` are presented as independent baselines that exemplify different aspects of this landscape: `DronWO` foregrounds embodied and writable interaction, while `TypeFly` demonstrates how LLM-based reasoning can structure autonomous behavior. Together, they reveal complementary strengths and limitations that motivate the development of the solution proposed in Chapter 4.

2.5.2. DronWO: A Writable HDI Baseline

`DronWO (Writable-Orthogonal)` [13] is a baseline system developed to explore how drone behavior can be *written* rather than programmed, allowing non-expert users to shape interaction through a natural user interface. The system introduces a conceptual shift: instead of viewing drone control as the execution of predefined commands, `DronWO` treats interaction as a writable process in which users actively construct the building blocks of behavior during use. This perspective responds to a recurring limitation in HDI systems, which often focus on a single aspect of interaction such as planning, gesture recognition, or user experience, without providing a framework in which users can meaningfully extend or reconfigure the system themselves.

At the core of `DronWO` is an *entity-based* architecture. Everything the drone can use or reason about is represented as an *entity*, organized into structured groups. Three overarching categories define the system: Real-World Entities, which represent objects or people, Abstract Entities, such as trajectories or numerical parameters, and Command Entities, which define the drone's actions. Entity groups may be either *fixed*, defined at design time, or *extensible*, allowing users to add new elements at runtime. This design implements the principle of *orthogonality*: entities from different groups can be freely combined through configurable commands. For example, a search command can be applied to different objects, trajectories, or time intervals simply by composing entities selected by the user.

Through extensible groups, users can "write" new trajectories or update lists of authorized individuals, effectively shaping the behavior space of the drone during interaction. This architecture creates a *writable interaction loop*. Users perceive how the drone re-

sponds, introduce new entities or adjust existing ones, and immediately see the effect of these modifications in subsequent behavior. In this sense, `DronWO` resonates with somaesthetic perspectives discussed in Section 2.3, since interaction becomes a process of bodily exploration in which users experiment, refine, and internalize their expressive patterns.

Empirical evaluations with non-expert participants confirmed the approach's accessibility. Users rapidly learned to compose commands and create personalized trajectories, and all participants successfully completed the assigned tasks. At the same time, the study revealed limitations. Cognitive load increased during complex compositions, drone feedback was not always sufficiently informative, and the system lacked semantic understanding of user intent beyond the combinatorial structure of its entities. Its behavior remained reactive and context-agnostic, with no capacity for long-term adaptation or reasoning about tasks.

`DronWO`'s value therefore lies not in its performance as an autonomous system, but in the conceptual space it opens. By foregrounding writability, orthogonality, and extensibility, it exposes a central question for HDI: how can interaction remain open and expressive while still supporting context awareness, safety, and reasoning? `DronWO` thus serves as an independent baseline in this thesis, illustrating the expressive and authorable end of the HDI spectrum against which more cognitively driven architectures, such as `TypeFly`, can be interpreted.

2.5.3. TypeFly: A Cognitive HDI Baseline

`TypeFly` [19] serves in this thesis as a cognitive baseline for HDI, demonstrating how LLMs can mediate goal-oriented collaboration between humans and aerial robots. Unlike embodied systems that focus primarily on gesture expressivity or writable interaction, `TypeFly` centers its design on reasoning, using a language model to interpret user intent and convert natural-language instructions into structured, executable plans.

At its core, `TypeFly` integrates an LLM as a decision-making engine. When a user provides a command such as "inspect the table on the right" or "fly around the chair and return to me," the system parses this instruction, infers spatial references, and produces a structured plan that decomposes the user's goal into actionable steps. This turns interaction into a conversational planning process in which the drone explains or clarifies its reasoning and the user remains in the loop.

`TypeFly` complements this reasoning capability with a lightweight perception pipeline that connects language-driven plans to what the drone can detect in the scene. The system identifies objects through vision modules and exposes them to the planner as symbolic

references that influence plan generation and determine which branch of the generated plan is executed. Object detections are used at decision time to guide action selection, rather than to maintain a persistent representation of the environment. Instead of constructing or updating a geometric model of the explored area, `TypeFly` treats perception as transient evidence that anchors individual steps to currently visible entities such as "table" or "chair." Consequently, the system does not accumulate spatial knowledge across time: detections support immediate execution but do not contribute to a lasting map or structural understanding of the scene. Moreover, `TypeFly` does not include a long-term memory mechanism; reasoning is performed on a per-task basis using only the information available in the current scene and dialog context. The system therefore cannot accumulate experience across sessions, retain user preferences, or refine its behavior over time, which motivates the memory-oriented approach proposed in this thesis. Contextual awareness depends entirely on what is recognized at the moment of planning, while safe execution relies on predefined flight boundaries and human supervision.

The complete architecture and evaluation of `TypeFly` are presented in Chapter 3. For the purpose of the present discussion, `TypeFly` represents a solid reference point for cognitive HDI: it demonstrates how multimodal perception and LLM-driven reasoning can be combined into a goal-oriented planning framework while simultaneously highlighting the absence of durable memory, deeper contextual grounding, and adaptive autonomy.

2.6. Summary and Synthesis

This chapter traced the evolution of HDI from early control-centric approaches to emerging forms of cognitive and reflective autonomy. Across this progression, HDI has moved from gesture-based interfaces and teleoperation to multimodal communication, semantic reasoning, and adaptive collaboration supported by LLMs. Table 2.1 summarizes the main paradigms and representative works discussed throughout the chapter, highlighting how each stage contributes to the broader trajectory of the field.

2.6.1. From Control to Cognitive Collaboration

Three consolidated paradigms characterize current HDI research, as summarized in Table 2.1. The first concerns manual and embodied control, where gestures and proxemics provide intuitive interaction but limited contextual awareness. The second introduces shared autonomy through multimodal inputs and safety logic, enabling more cooperative task execution. The third centers on cognitive adaptation via LLM- and VLA-based systems that couple linguistic reasoning with perception and action.

Paradigm / Direction	Core Characteristics	Representative Works
Manual Control	Direct gesture/body mapping.	Abtahi et al. [1], Cauchard et al. [15], Gio et al. [30], Natarajan et al. [46], Stoica et al. [52]
Shared Autonomy	Multimodal interaction with semi-autonomous behaviors, safety checks, and mixed initiative.	Cacace et al. [12], Medeiros et al. [45]
Cognitive Architecture	LLM- and VLA-based reasoning; natural-language goal interpretation and flexible task decomposition.	Ahn et al. [3], Driess et al. [24], Liang et al. [41], Lykov et al. [42], Team et al. [55], Zitkovich et al. [63], `TypeFly` [19]
Writable and Adaptive HDI	Emerging direction focused on interfaces that evolve during use, either through user-authored extensions or through interaction with and exploration of the environment; emphasizing systems whose behavior can change over time rather than remain fixed at design time.	`DronWO` [13], Ravichandran et al. [48]

Table 2.1: Summary of HDI paradigms and emerging directions.

A fourth direction is beginning to emerge in recent work. Rather than focusing solely on higher-level reasoning, this direction explores HDI interfaces that evolve during use, either through user-authored extensions or through interaction with and exploration of the environment. These approaches move beyond static interaction models toward systems whose behavior can change over time, reflecting both user practices and contextual experience. Although not yet an established paradigm, this growing interest defines the conceptual space in which the present thesis positions its contribution.

2.6.2. Positioning DronWO and TypeFly

Within this broader trajectory, `DronWO` [13] and `TypeFly` [19] serve as complementary reference systems that mark two different approaches to HDI. `DronWO` investigates writable, body-based interaction in which users compose behaviors by combining entities during

use. While this enables expressive authorship, it also introduces cognitive overload during complex compositions and relies on a rigid interface that cannot evolve unless new features are manually added by a programmer.

`TypeFly` addresses these issues through an LLM-driven planning pipeline that interprets natural-language goals and generates structured action plans, reducing user burden and increasing flexibility. However, it does not incorporate the writable and user-extensible qualities of `DronWO`, and it still operates without long-term memory or mechanisms for evolving its behavior over time.

Taken together, these systems highlight two essential yet incomplete perspectives on HDI: the expressive but rigid writability of `DronWO` and the flexible but non-authorable reasoning of `TypeFly`. Their limitations motivate the approach developed in the next chapter, which aims to bridge these capabilities by integrating reasoning, contextual grounding, and user-driven extension within a unified framework.

3 | TypeFly

In this chapter, we introduce `TypeFly` [19], a state-of-the-art open-source system that serves as the foundation of our work. `TypeFly` enables drones to execute tasks expressed in natural language by leveraging LLMs. In Section 3.1 we provide an overview of the system, while Section 3.2 describes its architecture, and Section 3.3 details the design of its main components, following the interaction pipeline from natural language input to task execution.

3.1. Overview

`TypeFly` is designed to bridge the gap between user instructions expressed in natural language and the low-level operations required by a drone. A user, who can be either a human or a language agent, provides a high-level *task description* in natural language. The system relies on an LLM to interpret this task in combination with contextual information from the environment and a description of the robot's capabilities. The output of this process is an *execution plan*, which is a structured sequence of actions encoded in `MiniSpec`, a lightweight domain-specific language specifically designed for robotic planning, which we describe in detail in Section 3.3. The plan is then validated and executed by the drone. For example, given the task "Find an apple", the system generates a MiniSpec plan that searches the environment for the specified object and directs the drone to approach it.

In what follows, the system is described from an interaction perspective: starting with the user's task description in natural language, then moving through context representation, plan generation, and finally the execution of the plan by the drone.

3.2. System Architecture

The design of `TypeFly` is based on three interacting elements, shown in Figure 3.1:

- a **drone equipped with a camera** (left side of Figure 3.1), which streams images

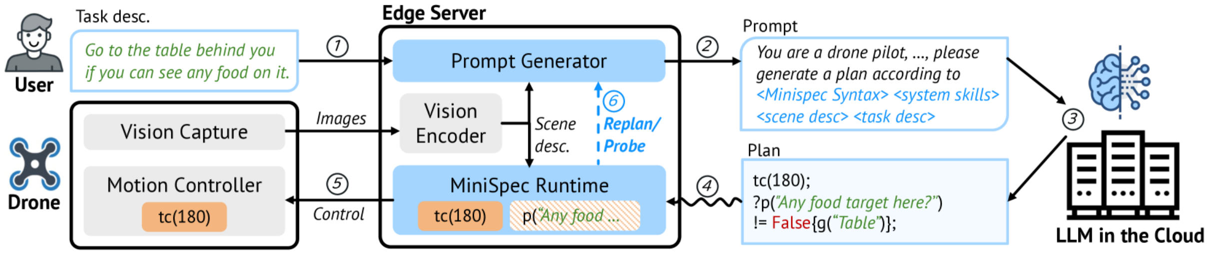


Figure 3.1: Overview of `TypeFly` [19]. (1) The user provides a natural-language task description. On the edge server, the `Prompt Generator` combines the task with a scene description produced by the `Vision Encoder` and with the available system skills. (2-3) The resulting prompt is sent to a cloud-based LLM, which returns a structured execution plan encoded in `MiniSpec`. (4) The plan is executed by the `MiniSpec Runtime`, which may trigger replanning or probing based on runtime feedback (6). (5) Control commands are sent to the drone's motion controller, while visual input is continuously captured to update the scene description.

of the environment to the edge server over a wireless connection and represents the final point of the pipeline where validated plans are executed through physical actions;

- an **Edge Server** (center of Figure 3.1), which locally processes the video stream to produce textual scene descriptions through `Vision Encoder`. It composes the LLM prompt by combining the task description, the current scene description, and the catalog of robot skills through `Prompt Generator` (steps 1-2). In addition, the edge server is also responsible for validating and executing the execution plans through `MiniSpec Runtime` before issuing commands to the drone (steps 4-6);
- a **Remote LLM** (right side of Figure 3.1), accessed as a cloud service, which provides high-level reasoning by turning the composed prompt (steps 2-3) into a structured execution plan expressed in MiniSpec.

This architecture cleanly separates sensing and actuation on the drone, local perception and plan interpretation on the edge, and abstract reasoning in the cloud, enabling `TypeFly` to transform natural language tasks into executable behaviors.

3.3. System Components

3.3.1. MiniSpec

The output of the LLM is expressed in `MiniSpec`, a lightweight domain-specific language (DSL) specifically designed for robotic planning. A DSL is a programming language created for a narrow application domain, in contrast to general-purpose languages such as Python or C. Its design is intentionally constrained: instead of supporting arbitrary programming constructs, it provides a small vocabulary tailored to the needs of the target domain. This specialization makes it easier to generate, verify, and execute code in that domain, at the cost of flexibility.

The choice of introducing a dedicated DSL in `TypeFly` is motivated by three design goals:

- **Conciseness**: MiniSpec syntax is compact, minimizing the number of tokens the LLM must generate. In the context of LLMs, a *token* is the basic unit of text processed by the model, typically corresponding to a short word, part of a longer word, or a punctuation symbol. The length of the generated output is therefore measured in tokens, and longer outputs require more computation and increase the computational cost of using a Remote LLM service. By reducing the number of tokens needed to express a plan, MiniSpec lowers both latency and cost, while keeping execution plans short and efficient.
- **Reliability**: MiniSpec restricts the available syntax to robot skills and a small set of flow-control elements, such as conditionals and bounded loops. This prevents the LLM from producing invalid or ambiguous code.
- **Safety**: By supporting only bounded constructs, MiniSpec ensures that every plan terminates within a finite number of steps. This eliminates the risk of infinite loops and avoids the dangers of executing arbitrary Python code, which could otherwise compromise the system.

In this way, MiniSpec acts as a safe and predictable intermediate representation, bridging the gap between natural language tasks and the low-level execution performed by the drone.

3.3.2. User Task in Natural Language

The interaction begins with the user providing a task in natural language (Figure 3.1, step 1). This is the starting point of the pipeline, where the system interprets the user's goal expressed informally and possibly ambiguously. Designing the system around natural language as input makes interaction intuitive and accessible, but also introduces challenges

such as vagueness, incomplete specifications, or multiple possible interpretations. The responsibility of the rest of the architecture is to transform this high-level, human-centric request into a concrete execution plan that is precise, unambiguous, and feasible for the drone.

3.3.3. Context Representation

To generate meaningful execution plans, the system must take into account the environment in which the drone operates. A key design choice of `TypeFly` is not to transmit raw images to the LLM, which would be inefficient in terms of bandwidth, slow to process, and potentially problematic for privacy. Instead, the video stream captured by the drone's camera is sent to the edge server, where it is processed locally to produce *scene descriptions* in natural language.

This processing is performed by the Vision Encoder, which relies on the YOLO real-time object detection model. YOLO analyzes each video frame and translates it into a structured list of the visible objects together with their properties, such as position within the image ((x,y) coordinates) and size (height and width). The result is a compact, structured summary of the scene that is easy for the LLM to reason about.

3.3.4. Robot Capabilities

Execution plans must be feasible given the physical limitations of the drone. To guarantee this, `TypeFly` exposes the robot's capabilities to the LLM in the form of *robot skills*, each documented in natural language and included in the planning prompt (Figure 3.1, step 2). Each skill has a name, a short natural language description of what it does, and an argument schema that specifies the types of its arguments. In a MiniSpec plan, each robot skill represents a unit of action, invoked in the same way as a function call in an imperative programming language.

The skills are structured hierarchically:

- **Low-level skills**: primitive operations that directly reflect the hardware's functionality and constraints. Examples include drone control primitives (e.g., move forward, turn clockwise), vision tools (e.g., obtain an object's location or dimension), user interface actions (e.g., provide textual feedback to the user), and the special `query` skill, which allows the system to engage the Remote LLM during execution. Their purpose is to provide complete coverage of what the drone can physically do. Assuming that all robotic platforms are equipped with a camera, the

only platform-specific low-level skills are those directly tied to drone control, which makes the design portable to other robotic systems.

- **High-level skills**: reusable abstractions defined by composing low-level or other high-level skills. For example, `TypeFly` provides the high-level skill `scan(object_name)`, which (i) exploits the YOLO-derived list of visible objects (see Section 3.3.3) to check whether `object_name` is currently in view; (ii) if not found, rotates the drone to explore other directions; and (iii) when found, commands the drone to approach the object. High-level skills are defined by the developer and provide direct leverage over the planning process. In this case, the LLM is encouraged to call `scan` to fulfill object-search tasks, rather than generating a sequence of low-level skills that may result in suboptimal behaviors such as checking only the frontal view or rotating without approaching the object. By introducing such abstractions, the developer can guide the LLM's reasoning, since the prompt design explicitly biases the model toward favoring high-level skills. Then, these abstractions shape the drone's behavior, steering execution toward reliable patterns and strategies anticipated by the developer.

Low-level skills constitute the entire capability of `TypeFly`, and in principle, the Remote LLM could produce plans based only on them. However, such plans are typically verbose and incur higher token production, making them less cost-efficient. On the other extreme, providing too many high-level skills would enlarge the planning prompt, which is problematic because LLMs have a fixed-length context window. Exceeding this window forces the model to truncate or ignore parts of the input, which can lead to the omission of relevant information, degraded reasoning quality, and ultimately incorrect or incomplete plans. This also increases the chance that the LLM overuses these abstractions and produces unreliable behaviors. Balancing between these two extremes, `TypeFly` includes a small but carefully chosen set of high-level skills that are widely applicable to common tasks, thereby simplifying plans and improving their quality with little overhead in the planning prompt.

Beyond the distinction between low- and high-level, the design of each skill follows a few guiding principles that make it easier for the LLM to invoke them correctly:

- **Easy argument passing**: skills are defined with as few arguments as possible, and these arguments are chosen so that their values can be directly extracted from the user task or the scene description. For example, a skill like `move_forward( distance_in_meter: float)` is preferable to a more general `move(x: float, y: float, z: float)`, since the former requires only one argument that can

often be directly extracted from the task description.

- **Provide options, not ranges**: whenever possible, arguments are categorical rather than numerical. For instance, setting speed through `set_speed(speed: str)` with options [slow, medium, fast] is more reliable than using `set_speed(value: int)` with a range from 0 to 100. Reliability here refers to the fact that categorical arguments reduce the risk of the LLM generating invalid or nonsensical values, since the model can directly match the options it has seen in training, rather than having to infer a precise number. This leverages the LLM's strength in pattern recognition and avoids errors due to arithmetic reasoning, which is not a strong suit of LLMs.
- **Alignment with available context**: when numerical arguments are unavoidable, they are designed to align with information already present in the scene description. For example, a navigation skill would be designed as `move_to(location)` rather than `move_by(displacement)`, because object locations are included in the scene description, whereas displacements would require additional calculation.

These principles reduce the cognitive load on the LLM and improve the robustness of generated plans, ensuring that the available skills can be used effectively without introducing unnecessary complexity.

3.3.5. Prompt Generator and MiniSpec Runtime

The `Prompt Generator` orchestrates the planning process. Its role is to combine the three essential inputs: the user's task, the scene description, and the robot skills. These elements are merged into a carefully designed prompt for the LLM (Figure 3.1, step 2). The prompt is engineered to encourage the use of high-level skills, which leads to plans that are more concise and easier to interpret, while also steering the reasoning process according to abstractions introduced by the developer.

The output of this process is a candidate execution plan expressed in MiniSpec. At this stage, the plan must be validated and linked to the robot's actual capabilities. This role is handled by the `MiniSpec Runtime` (Figure 3.1, step 5), which checks the validity of the generated plan, ensures that only available skills are invoked, and translates the symbolic instructions into concrete actions.

Together, the `Prompt Generator` and the `MiniSpec Runtime` ensure that only executable and safe plans reach the execution stage, providing robustness in the transition from symbolic reasoning to physical behavior.

3.3.6. Prompt Engineering

A central feature of `TypeFly`'s design is the careful construction of prompts that guide the LLM during planning. Prompt engineering serves two purposes: it increases the likelihood of producing syntactically valid MiniSpec code, and it steers the LLM toward efficient and semantically correct plans.

To make the MiniSpec syntax available to the model, the entire grammar is explicitly included in the planning prompt. The grammar is written in a BNF-like notation, which concisely specifies the allowable constructs of the language. In this section we report only a representative subset, to illustrate the form in which the grammar is provided:

```
<program> ::= { <block-statement> [';'] | <statement> ';' }
<statement> ::= <variable-assign> | <function-call> | <return>
<function-call> ::= <function-name> ['(' <argument> ')']
```

This excerpt shows the general structure: a program consists of statements, which can include function calls, variable assignments, or returns. By embedding the complete grammar in the prompt, the LLM is explicitly constrained to generate plans that conform to MiniSpec syntax.

`TypeFly` employs several complementary strategies:

- **Reasoning guides**: the planning prompt embeds explicit instructions that shape the reasoning process of the LLM. These guides emphasize the connection between task and scene descriptions, instructing the LLM to ignore irrelevant details from the current scene when they do not pertain to the user's task. They also regulate the use of the query mechanism: a query should only be invoked when the information required is not already available in the current scene description. By embedding these rules into the planning prompt, the LLM is less prone to common mistakes and better aligned with the execution context.

- **Few-shot examples**: the planning prompt includes a small but representative set of example plans that demonstrate how system skills can be used. These examples showcase both low-level and high-level skills and cover a broad range of tasks. Leveraging the few-shot learning ability of LLMs, this strategy increases the chance that the generated plan combines skills correctly and generalizes to unseen tasks.

- **Goal-specific prompts**: `TypeFly` does not rely on a single static prompt, but uses distinct prompt templates depending on the current goal. For instance, when generating an execution plan, the prompt focuses on task reasoning and the structured

use of skills, whereas during execution the system employs a dedicated query prompt with different guides and specific usage examples. This separation ensures that the Remote LLM is provided only with the information and instructions relevant to the current phase, improving both efficiency and correctness.

Together, these strategies make prompt engineering an integral part of the system design. They not only improve the syntactic validity of MiniSpec programs but also increase semantic correctness, enabling `TypeFly` to generate plans that are both executable and efficient.

3.3.7. Interaction During Execution

Not all the information needed for task execution can be obtained upfront. Real-world environments evolve over time, so the situation at execution may differ from the one observed during planning. Moreover, some aspects of the environment require additional reasoning that cannot be anticipated.

To address such cases, `TypeFly` allows the plan to request extra information during execution. One option is to interact with the Remote LLM through a dedicated `query` mechanism. When a plan issues a query, the system composes a focused prompt that includes the latest scene description together with a specific question. The reply from the LLM is immediately integrated into the ongoing execution, steering subsequent actions. This mechanism is powerful because the LLM can perform abstract reasoning, such as deciding whether an object is edible or suitable for a given task.

A second and simpler option is to query the Vision Encoder directly. In this case, the plan does not ask the LLM to reason, but instead checks factual properties of the current scene, for example whether a specific object is visible. Because this approach only requires local processing of the YOLO-generated object list, it adds very little overhead. Compared to sending a request to the remote LLM and waiting for the response to be generated and transmitted back, this solution is both faster and computationally cheaper, and is sufficient whenever the required information is already contained in the scene description.

The two mechanisms therefore complement each other: queries to the Vision Encoder are lightweight and efficient for verifying observable facts, while queries to the Remote LLM are necessary when higher-level reasoning about abstract properties is required.

As an example, consider the user task *"find something edible"*. Since the property *edible* is abstract and cannot be derived from raw visual information alone, the plan must query the Remote LLM. In contrast, if the task is *"find an apple"*, the plan can simply query

the Vision Encoder to check whether the object appears in the latest scene description, avoiding the overhead of involving the LLM.

4 | Design of an Adaptive, Context-Aware Interface

This chapter presents `DACS`, an evolution of the `TypeFly` framework introduced in Chapter 3. The name stands for `Drone Adaptive Context-aware System`, emphasizing two central principles: *adaptive interaction* and *context awareness*. The system is designed to enable drones that can perceive their surroundings, learn from human interaction, and progressively refine their behavior through experience.

Leveraging insights from the writable interfaces explored with `DronWo` in Section 2.5.2, our work investigates how natural language interactions, powered by LLMs, can evolve over time. `DACS` extends this idea by introducing new mechanisms for high-level skill creation and memory of past interactions. Combined with reasoning over environmental information, these capabilities allow the system to adapt to new users, environments, and tasks without the need for explicit reprogramming, thereby transforming static command execution into a continuous, mutual learning process.

The chapter is organized as follows. Section 4.1 outlines the main limitations of the baseline `TypeFly` framework and the motivation for a more adaptive solution. Section 4.2 introduces an illustrative example used throughout the chapter to clarify the differences between the baseline and the proposed system. Section 4.3 details the architecture of `DACS` and its main components, including the conceptual model, the autonomous creation of high-level skills, the memory and the use of visual information for environment-aware reasoning. This section also describes mechanisms for user clarification in ambiguous tasks and for defining natural-language flyzones, both of which enhance the adaptability and the safety of the interaction. Altogether, these elements illustrate how `DACS` turns the static pipeline of `TypeFly` into an adaptive, context-aware framework for natural human–drone collaboration.

4.1. Design Motivation

This section introduces the motivation and guiding principles behind `DACS`. LLMs are increasingly considered for UAV control, as highlighted by recent surveys that identify both their potential and the open challenges of integrating adaptability, learning, and environmental reasoning into real systems [35]. Building on these observations, Section 4.1.1 revisits the main shortcomings of existing paradigms and of the baselines `TypeFly` and `DronWO`, while Subsection 4.1.2 outlines how `DACS` addresses them through mechanisms for memory, feedback, and contextual awareness.

4.1.1. Limitations of Existing Solutions

It is important to examine the limitations of existing approaches to drone interaction. We first outline the weaknesses of the two dominant paradigms, namely the *controller-like mode* and the *semi-autonomous mode*, and then focus on the two systems that are most relevant to our work: `DronWo` and `TypeFly`.

General limitations of controller-like and semi-autonomous modes. Despite their widespread use, these paradigms share three central limitations:

- **Rigidity and lack of personalization.** Interfaces are predefined at design time and users have little or no possibility to extend or modify them. This rigidity prevents adaptation to novel tasks that may arise in everyday contexts and hinders the discovery of new, user-driven behaviors.
- **Lack of learning over time.** Interactions are treated as isolated events. When a command is repeated, the response remains identical, without incorporating past user feedback, preferences, or contextual changes. As a result, the system does not evolve together with the user or the environment.
- **Exclusivity of use.** Effective control often requires prior expertise, either knowledge of drone dynamics or training with a specialized interface. Non-expert users typically struggle to achieve even simple tasks without support, reducing accessibility.

DronWo and the Drone Gymnasium. `DronWo` introduces a *writable interface* that partially addresses these issues. It allows users to define new gesture-action mappings at runtime. For example, a user can record a trajectory, bind it to a gesture, and later reuse it to search for an object. This opened the door to personalization without requiring

programming skills, while still keeping a structured interaction model.

The potential and the limitations of `DronWo` emerged clearly during the **Digital Futures Drone Gymnasium**, a workshop and exhibition held at KTH in Stockholm in September 2025.[1] The event was organized by Professors Luca Mottola (Mobile Robotics, Politecnico di Milano) and Kristina Höök (Interaction Design, KTH), in collaboration with the Swedish company Bitcraze, makers of the Crazyflie nano-quadcopter.[2] The Drone Gymnasium was conceived as a living lab, a hybrid between a technical testbed and an experimental playground, where researchers, students, and practitioners could collectively explore the social, embodied, and technical dimensions of drones as companions and co-actors in shared spaces.

Its objective was to investigate how drones can move beyond the role of tools to become part of everyday environments. The main questions were: how does it feel to share space with a flying robot? What makes a drone feel intrusive instead of supportive? How can interaction be distributed across the body rather than concentrated in a controller?

Two main patterns of user engagement emerged:

- A *feature-composition style*, where participants explored `DronWo`'s writable interface by combining gestures, trajectories, and object mappings to invent playful mini-games.
- A *material-exploration style*, where participants treated drones as physical artifacts, attaching strings or loads to test collisions, carrying behavior, or new tool-like extensions.

These explorations highlighted both the strengths and the shortcomings of `DronWo`:

- **Cognitive burden.** Users had to memorize an expanding set of gesture-action mappings, including the ones they defined themselves. In fact, we provided students with a cheat sheet listing all predefined gestures, but for user-defined mappings they were on their own, which increased the memory load over time.
- **Functional ceiling.** `DronWo`'s autonomy is constrained to predefined primitives such as trajectory recording or object search. In fact, during the Drone Gymnasium we organized prototyping days to add new functions, such as reducing the duration of recorded trajectories and making them composable, but any further extension still required direct developer intervention.

[1] https://www.digitalfutures.kth.se/project/the-digital-futures-drone-gymnasium/
[2] https://www.bitcraze.io/2025/09/making-social-robots-fly-inside-the-drone-gymnasium/

- **Limited context-awareness.** The system responded only to user gestures and to objects shown to the camera as tokens to be mapped to specific points or actions. In fact, the drone could not capture the meaning of one object versus another, nor situate them in a broader context, for example distinguishing between a kitchen object and an office object. This limited environmental understanding.
- **Session continuity.** Initially, mappings disappeared after each session, forcing users to redefine them. In fact, we later added persistence to store user-defined mappings across sessions, but the system was still not able to interpret user feedback, adapt to preferences, or evolve its behavior over time.

TypeFly: natural language as an alternative. In parallel, `TypeFly` offered a different baseline. Its core contribution was to reduce memorization costs by introducing natural-language interaction: users could simply issue spoken or written instructions, which the system translated into executable drone plans. This approach effectively mitigates the cognitive load observed in `DronWo`, making the system more accessible to novices.

However, `TypeFly` also exhibits limitations:

- It does not learn over time. User preferences and behaviors are not stored or reused across sessions.
- Its perception of the environment is limited. The LLM receives only a textual list of detected objects from the visual encoder, without spatial or contextual reasoning. For example, it cannot recognize that the drone is in a kitchen or in an office. This constrains its ability to generate context-sensitive plans.

Motivation for DACS. Taken together, `DronWo` and `TypeFly` illustrate a continuum of trade-offs. `DronWo` empowers personalization but imposes memorization costs and reaches a functional ceiling. `TypeFly` reduces cognitive effort but lacks adaptation and environmental grounding. Neither fully integrates user experience with contextual awareness.

These complementary gaps motivated the design of `DACS`, an adaptive and context-aware system that combines natural-language reasoning with long-term learning and richer environmental perception.

4.1.2. Solution Overview

`DACS` is designed to overcome the limitations identified above by introducing mechanisms that enable context awareness and learning from the past. Its design stems from a guiding

idea: the interaction between humans and drones should not remain fixed, but evolve over time, shaped by both the environment and the user's behavior. Rather than simply interpreting natural language commands, the system continuously learns from experience, refines its understanding of user intent, and adapts its reasoning to contextual cues.

The architecture of `DACS` extends the baseline `TypeFly` framework through five core innovations that collectively transform the interaction model into a dynamic, self-improving process:

- **High-level skill creation.** In contrast to the baseline `TypeFly`, where high-level skills are exclusively defined in advance by the programmer, `DACS` enables the system to autonomously generate new high-level skills through the reasoning capabilities of the LLM. During task execution, the LLM can combine existing skills to create new reusable high-level skills.

  These pre-packaged behaviors allow the system to adapt its behavior over time, reducing the reasoning effort required by the LLM to generate plans for recurring tasks and ensuring more consistent execution. Instead of generating an entire plan from scratch for similar user requests, the LLM can reuse an existing high-level skill, i.e., a previously defined behavioral module.

  This mechanism enables the system to evolve its action space and autonomously develop specialized skills tailored to user habits and specific goals. As a result, the user interface is progressively enriched with ready-to-use skills that integrate the most frequently required behaviors and are customized for each user. Consequently, each user's interface evolves differently over time, depending on individual usage patterns.

  This approach allows the system to exploit previous reasoning during current plan generation, both to improve performance and to better align generated plans with user expectations. In particular, if a previously defined skill has been validated through user feedback, as we will see in Section 4.3.4, it is likely to be suitable for identical or similar tasks in future executions. Further details on this functionality are presented in Section 4.3.3.

- **Multi-level memory.** The system integrates both short-term and long-term memory layers (Section 4.3.4). The *short-term memory* continuously records the current execution context, capturing the state of plan execution across iterations after re-planning and the outcomes of executed skills. This information helps the LLM plan subsequent iterations more effectively by considering what has already been

accomplished. The *long-term memory* includes several complementary components: user feedback, shortcuts, and environmental memory. These mechanisms together maintain user preferences, store validated task executions for rapid reuse, and build a persistent, graph-based representation of the environment. This structure allows the system to recall and reuse knowledge from past experiences, reason about prior results, and progressively adapt its decisions and behavior over time.

- **Visual environment awareness.** `DACS` enables the LLM to reason not only over textual descriptions of the environment but also over visual information captured from the drone's onboard FPV (first-person view) camera, exploiting the multimodal capabilities of modern LLMs. By integrating FPV visual perception with textual reasoning, the system allows the LLM to understand the drone's current position, what it observes from its viewpoint, and how to act according to the perceived visual scene. This tight coupling between perception and reasoning supports context-aware decision-making. A detailed explanation of this mechanism is provided in Section 4.3.5.

- **User-guided adaptation.** When a task is ambiguous or incomplete, the system proactively requests clarifications from the user before generating a plan. Furthermore, during execution, the LLM can also request additional information whenever ambiguities arise, ensuring that the task remains aligned with the user's intent throughout the entire process. The details of this feature are discussed in Section 4.3.6.

- **Flyzones.** `DACS` allows users to define flyzones, that are bounded operational areas, directly through natural language. These user defined descriptions are translated into spatial constraints that guide both planning and execution, ensuring that the drone operates only within regions relevant to the task.

  In indoor scenarios such as warehouses or industrial facilities, flyzones can be used to specify physical boundaries, for example aisles, storage areas, or restricted zones, effectively constraining the drone motion to a well defined workspace. This bounded context simplifies task execution, improves reliability, and prevents the drone from entering areas where operation is undesired or unsafe.

  In outdoor scenarios, flyzones are particularly useful in emergency and disaster response settings. For instance, during search and rescue operations following a natural catastrophe, responders can specify geographic boundaries that focus the drone search on critical areas. By limiting exploration to these regions, the system increases search efficiency and ensures that resources are allocated where they are

most needed.

Additional details on the definition, management, and impact of flyzones are presented in Section 4.3.7.

Together, these components redefine the interaction paradigm from a static pipeline into a system that evolves through experience. By combining natural language understanding, visual perception, and multi-level memory, `DACS` establishes a more fluid and context-aware form of collaboration between human and drone, capable of growing more effective with each use.

4.2. Illustrative Example: *Find an Apple*

To illustrate the evolution introduced by `DACS`, we consider a simple yet representative task: a user instructs the drone to *"Find an apple"*. This scenario highlights how the proposed system builds upon the baseline `TypeFly` to achieve a more adaptive and context-aware behavior.

In the baseline `TypeFly`, the instruction is processed as a natural language command that the system translates into a MiniSpec plan. In such a case, the generated plan typically consists of a predefined search strategy in which the drone scans its surroundings by performing successive 45-degree rotations on the horizontal plane, leveraging the predefined `scan` high-level skill. After each rotation, the onboard camera captures an image of the environment, which is analyzed by the Vision Encoder using the YOLO object detection model. The model outputs a list of visible objects in the current frame, each described by its label, position, and bounding box dimensions. A local checker then compares this symbolic scene description with the target class specified by the user, in this case an *apple*. If the object is not found, the plan continues the rotation; otherwise, if it is present, the plan terminates the scan and commands the drone to approach the detected object. However, if the apple is not detected after a full scanning routine, the baseline `TypeFly` triggers a replanning mechanism that generates a new plan equivalent to the previous one, again exploiting the predefined `scan` high-level skill. As a result, the drone remains stuck executing the same scanning routine without exploring alternative strategies. Although effective in structured environments, this process is static and context-independent: every instance of the same command results in an identical scanning routine, regardless of previous experience or environmental context. This limitation arises because, in `TypeFly`, the drone is not able to actively reason about its surrounding environment. Instead, it operates only on a flat list of detected objects, without abstracting higher-level contextual information. For example, it does not reason about the type of environment it is

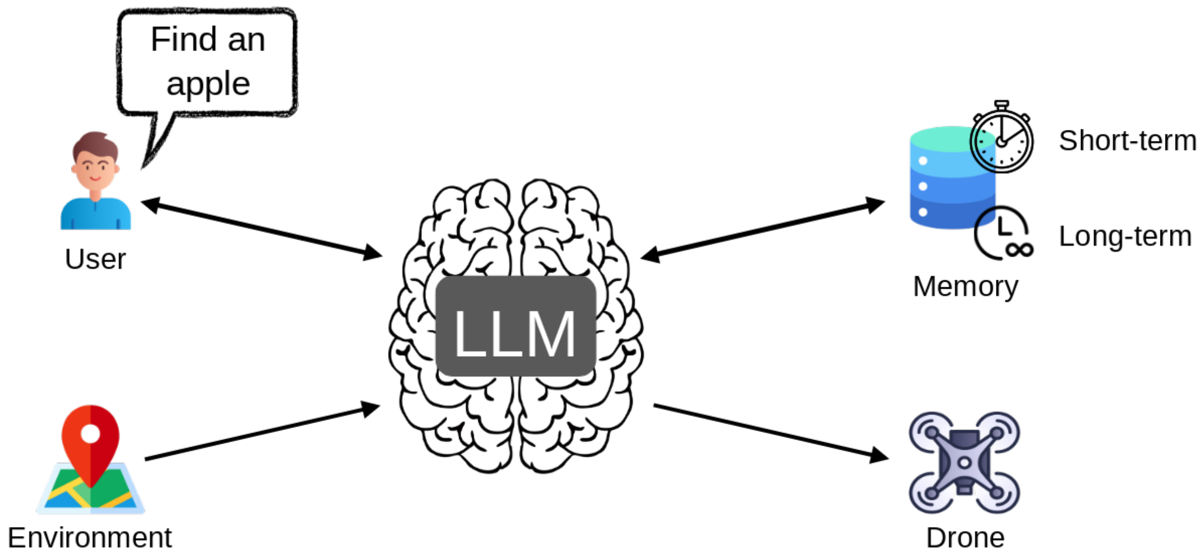


Figure 4.1: Conceptual model of DACS. The LLM acts as the central reasoning component, interacting with the user, memory, environment, and the drone, which executes the generated plans.

operating in or how environmental features and the presence of other objects could be exploited to accomplish the task. Moreover, the system does not retain information from past interactions, which prevents it from leveraging previous observations to inform future reasoning and task execution.

DACS enhances this interaction by introducing context awareness and memory. When receiving the same task, the system can exploit information gathered during previous missions, such as the location of regions where specific objects were detected. This knowledge enables it to prioritize regions that are more likely to contain the target, instead of starting from a blank search. If the apple is not immediately visible, instead of stopping, the drone actively searches the environment by exploiting contextual information and object relations. For example, the presence of other fruits can guide the drone to explore a particular direction and initiate a new scanning routine in that area. Moreover, the system can dynamically adapt its scanning behavior based on previous user feedback, such as "notify me when you find the target object", and can request clarification from the user when necessary in ambiguous situations.

This example demonstrates how contextual understanding and adaptation to user needs can transform a fixed control routine into a more flexible and intelligent behavior. While TypeFly follows a predefined scanning pattern and gets stuck if the target is not detected, DACS actively builds on experience, user feedback, and environmental cues to refine its reasoning and improve collaboration between humans and drones.

4.3. Our Solution - DACS

4.3.1. Conceptual Model

Before describing the individual components of `DACS`, it is useful to introduce its conceptual model, illustrated in Figure 4.1. At the center of the system lies the LLM, which represents the reasoning core, the “brain” of the drone. All interactions flow through this central component, which interprets information, generates plans, and adapts its behavior over time through feedback and experience.

The LLM interacts with four main elements of the system:

- **User.** The user expresses intentions through natural language, formulating high-level goals such as *“find an apple”*. The LLM interprets these requests, reformulates them into structured plans, and, in turn, communicates progress or clarifications back to the user. This bidirectional exchange allows the system to refine understanding, resolve ambiguities, and progressively align its behavior with user expectations.
- **Memory.** Memory serves as the adaptive substrate of the system. It accumulates knowledge about both the user and the environment, capturing habits, preferences, and contextual information derived from previous interactions. This persistent knowledge allows the LLM to generate plans that are increasingly coherent with the user’s style of interaction and the characteristics of the operational space.
- **Environment.** The drone continuously perceives the surrounding environment through its sensors and camera. This information is processed locally to produce symbolic and visual representations that the LLM can exploit to reason about the scene. By integrating live environmental data, such as the objects present and the visual context of the drone’s location, the system can make informed decisions, for example, about which direction to explore to efficiently complete a task.
- **Drone.** The drone represents the physical endpoint of the system’s reasoning. It receives validated plans and executes them in the real world through motion and perception actions. The outcomes of these actions are then relayed back to the system, closing the perception-action loop that sustains continuous adaptation.

This modular view resonates with the design of `CognitiveOS` [42], where large multimodal models coordinate specialized subsystems for perception, memory, and action. `DACS` adopts a similar principle, but tailors it to the needs of adaptive, context-aware UAV operation. Such a modular architecture also makes it straightforward to extend the

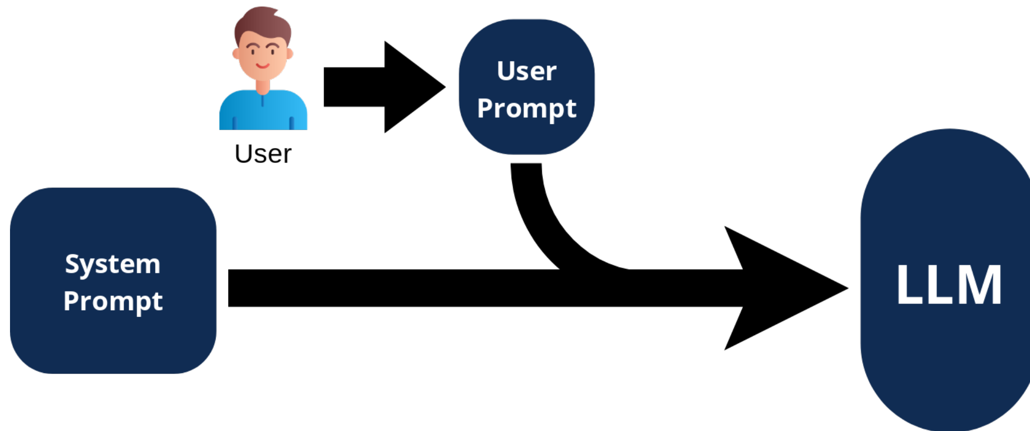


Figure 4.2: Dual-prompt interaction model in `DACS`. Each LLM instance is configured with a fixed system prompt that defines its role, while user instructions are provided separately through user prompts.

system with additional components when required, for example a dedicated module for enforcing ethical or safety rules, or for domain-specific reasoning in particular application contexts.

4.3.2. Prompting Strategy

In the baseline `TypeFly` framework, prompts are not explicitly separated, as no clear distinction exists between the user prompt and the system prompt. Each time a user interacts with the system, the entire context is provided in a single prompt that mixes system-level instructions with the user request. This approach has two main drawbacks. First, the role and responsibilities of the LLM are not clearly defined. Second, execution is slowed because the full set of instructions must be reprocessed for every new interaction.

`DACS` addresses these limitations by introducing a dual-prompt strategy based on specialized LLM instances, as illustrated in Figure 4.2. Each feature of the system, intended as a distinct functional capability that requires autonomous reasoning (such as planning, memory management, perception, or user feedback), is implemented by an independent *LLM instance*, where an LLM instance is defined as a language model paired with a fixed system prompt that determines its role and behavior. The interaction between the system prompt, the user prompt, and the LLM is depicted in Figure 4.2. For each feature implemented through an LLM instance, we define:

- **System prompt.** The system prompt is fixed and defines the role, scope, and reasoning strategy of the LLM instance assigned to a specific feature. It encodes how the LLM should interpret inputs, apply domain-specific logic, and produce

structured outputs.

- **User prompt.** The user prompt contains the actual instruction provided by the human operator and is interpreted within the context established by the system prompt. For example, if the system prompt defines an LLM instance as responsible for feedback handling, the corresponding user prompt might be "the task execution was good, but next time tell me what you are going to do".

This design ensures that each LLM instance has a well-defined responsibility and that interactions remain logically structured and modular. The separation between system and user prompts, visually summarized in Figure 4.2, provides two practical benefits:

- **Reduced latency.** Since system prompts are preloaded and remain unchanged, the LLM instance does not need to re-interpret them at every interaction. At runtime, only the user prompt varies, reducing planning latency.
- **Clear division of responsibilities.** System prompts define stable reasoning policies and constraints, while user prompts specify task-level intent. This separation prevents interference between system logic and user variability, and it enables modular updates to system behavior without affecting the user interaction layer.

In practice, most features of `DACS`, including plan generation, short-term memory, user feedback handling, visual environment awareness, and flyzones, are implemented through the combination of one system prompt and one user prompt. The system prompt establishes the reasoning environment of the corresponding LLM instance, while the user prompt injects task-specific intent at runtime.

4.3.3. High-Level Skill Creation

In `TypeFly`, high-level skills are defined exclusively by the programmer during system development, as illustrated in Section 3.3.4. These skills combine multiple low-level actions into reusable abstractions, but their repertoire remains fixed at runtime.

In `DACS`, this limitation is overcome by granting the LLM the ability to create new high-level skills dynamically. This design choice allows the system to grow its behavioral repertoire over time, developing skills that reflect the specific needs of its user. Whenever a new task is described in natural language, the LLM can encapsulate the generated plan into a named high-level skill, which can then be reused in future interactions. Over time, the system learns which skills are most relevant and retrieves them when appropriate, avoiding repeated long reasoning for tasks that share similar structure or intent. The decision of whether to create a new high-level skill, modify an existing one, or reuse a

previously defined skill is taken autonomously by the LLM and is governed by its system prompt, which encodes the criteria and policies for skill management.

Example: *Find an apple.* The benefits of this approach can be illustrated through the *"find an apple"* task introduced in Section 4.2. The LLM generates a plan that instructs the drone to rotate incrementally, analyze the YOLO-detected objects in each frame, and approach the target once identified. Recognizing the generality of this sequence, the system can encapsulate it into a reusable high-level skill named `search_object(object_name)`. From that point on, the skill can be invoked directly whenever a similar search task arises, such as `search_object("banana")` or `search_object("orange")`. This reuse reduces plan generation time and ensures consistent, tested behavior as explained above.

As a further example, consider the previous scenario where the user instructs the drone to *"create a skill that follows a person"*. Afterward, the user might issue a composite task such as *"Follow a person and then find an apple"*. The LLM quickly realizes that it can compose two existing high-level skills, `follow_person()` and `search_object("apple")`, executing them in sequence. This process, which in `TypeFly` would require generating a long, complex plan from scratch, is now simplified into a modular and reliable composition of previously validated behaviors.

Although this mechanism is not strictly essential, since the LLM could in principle generate plans solely from low-level skills, it provides two major advantages. First, it increases *efficiency*: when a reusable high-level skill already exists, the LLM can invoke it directly rather than generating a full plan composed of multiple low-level actions. This reduces reasoning time and lowers the likelihood of syntactic or semantic errors. Second, it improves *consistency*: recurrent user tasks are executed through behaviors that have already been validated by the user. When a task is first requested, the LLM generates a plan that is encapsulated into a high-level skill. If the user is satisfied with the execution, positive feedback confirms that this plan is effective. The next time the same task arises, the system reuses exactly the same high-level skill, ensuring the behavior remains unchanged. Without this mechanism, the LLM could generate a slightly different plan even after positive feedback, as each plan is regenerated from scratch rather than reused. Conversely, if the user provides negative feedback, the plan is revised as described in Section 4.3.4. The main drawback of this approach, which is the potential repetition of incorrect or suboptimal behaviors, is in fact mitigated through user feedback and by the possibility for the LLM of overwriting an already defined high-level skill.

High-level skills can also be created explicitly by the user through the interface. This interaction mode is primarily intended for expert users who are aware of the internal

functioning of the system and have received a short and simple training on how high-level skills are defined and used. When the user anticipates that a capability will be useful in the future, they can instruct the LLM to define it in advance through natural language. For example, a user might request: *"Create a skill called `follow_person` that tracks a person detected by the camera"*. The LLM then generates the corresponding MiniSpec code and stores it as a high-level skill. Users may also ask the system to explain how the skill has been implemented, gaining transparency and confidence in its subsequent use. Moreover, the user can request modifications to the implementation in an iterative process, where each time the LLM overwrites the same high-level skill until the desired behavior is achieved.

In summary, enabling high-level skill creation within the LLM transforms the system from a static executor into a self-expanding, user-aware assistant. By progressively building its own library of reusable abstractions, `DACS` achieves both greater efficiency and behavioral consistency, paving the way for long-term adaptability in human-drone interaction.

4.3.4. Memory

A key feature of `DACS` is the integration of a multi-level memory system that enables the drone to adapt its behavior over time. Unlike the baseline `TypeFly`, which interprets each user request independently, `DACS` retains information about previous interactions and environmental conditions. This allows the system to develop a richer understanding of both the user and the environment, ultimately producing more coherent and context-aware plans.

As illustrated in Figure 4.3, the memory is structured into two layers. On the left, the `Short-Term Memory` maintains a dynamic execution trace of the current task, while on the right the `Long-Term Memory` consolidates knowledge through three mechanisms: *user feedback*, *shortcuts*, and the *environmental memory*.

Short-Term Memory

A task is often fulfilled through multiple steps, where each step has its own generated plan that progressively moves closer to completing the overall objective. In this context, maintaining a memory of the task status is particularly helpful, since it enables the LLM to generate improved plans for subsequent steps by building on what has already been attempted. To address this need, the `Short-Term Memory` maintains an explicit record of the ongoing execution process, acting as a dynamic trace that evolves in parallel with the current task.

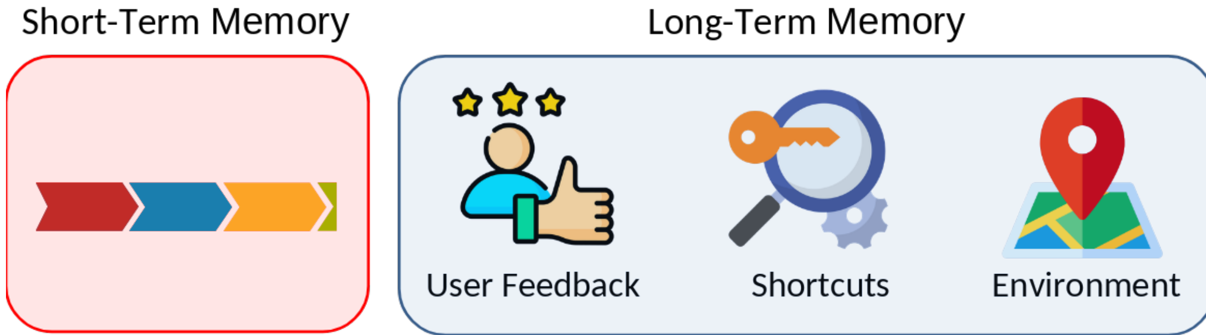


Figure 4.3: *Multi-level memory* architecture of DACS.

In the baseline TypeFly, an execution history is already present, but it stores only the MiniSpec code generated at each iteration. Consequently, the LLM has access only to the static program structure, without visibility into the intermediate execution outcomes produced at runtime or the specific branches taken depending on environmental conditions and observed results. As a result, the LLM lacks awareness of what actually occurred during each interaction step and cannot condition subsequent plans on the evolving execution context.

DACS augments this representation by attaching outcome-oriented annotations to every step. For each executed action, the system records what is attempted, what is observed, and whether the attempt is successful, partially successful, or unsuccessful. By layering this information on top of the plan, the LLM can condition the next step on the actual state of execution rather than on an idealized plan, thereby shaping subsequent actions in real time. This mechanism aligns with the concept of *active sensing* in the robotics literature, where perception and action are tightly coupled and sensing actions are deliberately selected based on their expected informational gain [4, 6].

This design also echoes the approach of ProgPrompt, where natural language instructions are translated into program-like representations that explicitly track state and context so that execution can adapt as new outcomes are observed [50]. In DACS, short-term memory plays a similar role by maintaining an evolving record of what has been attempted and what results have been obtained, enabling the LLM to refine subsequent plans based on up-to-date information.

Example: *Find an apple.* In TypeFly, the instruction *"find an apple"* triggers a predefined scanning routine that consists solely of in-place rotations, where the drone checks each direction for the presence of an apple. In this setting, short-term memory has little practical value, since tasks are executed from a fixed location. The drone does not

actively explore the environment but continues rotating until an apple is detected, potentially looping indefinitely if the object is not found. When no new objects appear in the immediate surroundings, iterating over multiple steps provides no additional information.

Moreover, at each step only the MiniSpec code of the generated plan is stored, as explained above. As a consequence, the system merely records that the drone executes a full rotation, while retaining no information about what occurs at runtime, such as the objects that are observed, which parts of the code are executed, or which branches are not taken. This behavior is closely related to the static nature of `TypeFly`: since the drone cannot actively move through the environment to gather new information, there is limited opportunity to exploit past observations. As a result, the reasoning capabilities of the LLM are not fully utilized.

In contrast, `DACS` actively exploits short-term memory by capturing informative outcomes at each step, for example "apple not found in the current view, moved to the right because a kitchen is identified". Given this annotated execution trace, the LLM leverages the accumulated information to adapt the plan, such as initiating exploration of the kitchen as a plausible region in which to locate an apple. This transformation of stepwise outcomes into guidance for subsequent actions illustrates how short-term memory enables meaningful, context-sensitive progression beyond a static scanning routine.

User Feedback

User feedback provides the foundation for adaptive learning in `DACS`. After each execution, the user can provide feedback describing what went well and what went wrong, allowing the system to preserve effective behaviors and revise or replace ineffective ones. Unlike `TypeFly`, where interaction remains static across executions, feedback in `DACS` persists over time and directly influences future planning decisions. These signals update preferences and constraints that guide the system's reasoning beyond the current task.

A key design choice in `DACS` is the distinction between *task-related feedback* and *universal feedback*. This distinction enables the user to shape the drone's behavior at different levels of granularity, ranging from specific task families to global interaction policies. At runtime, the LLM instance responsible for plan generation interprets the user's feedback and determines its scope based on its semantic content and contextual relevance.

Task-related feedback applies to a specific task or to a family of related tasks. For example, a user instruction such as *"Next time, when you find an object, rotate 360 degrees before stopping"* is interpreted as relevant to object-search behaviors. The LLM associates this feedback with the corresponding class of tasks and integrates it into the definition or

revision of the related high-level skills, as well as into subsequent plan generation. As a result, the modified behavior is automatically reused whenever similar tasks are executed, while remaining inactive for unrelated tasks.

Universal feedback, by contrast, expresses global preferences that affect the drone's behavior across all tasks. Instructions such as *"Tell me every step of the plan before executing it"* or *"Always keep a safe distance from people"* are interpreted as interaction-level or safety-level constraints. In these cases, the LLM updates global reasoning policies that are applied consistently during plan generation, regardless of the specific task being executed. This allows users to modify the overall interaction style of the drone.

By analyzing accumulated feedback, the system progressively refines its reasoning process across interactions. This contrasts with the baseline `TypeFly`, where feedback affects only the current execution and does not influence subsequent behavior, resulting in an interaction model that remains static over time.

Example: *Find an apple.* Suppose that, at the end of an execution, the user says: *"Next time, when you find the object I asked for, tell me how many of them are there"*. `DACS` interprets this as task-related feedback associated with object-finding behaviors. On subsequent executions of *"find an apple"* or similar tasks, the LLM synthesizes a revised high-level skill that extends the original plan with a counting-and-reporting step (e.g., detect apple → count instances → announce total).

This behavior highlights the dynamic nature of `DACS`. The interaction evolves over time because feedback is retained, generalized at the appropriate scope, and reused. By deciding whether feedback applies locally or globally, the system aligns its adaptations with the user's intent, supports consistent personalization, and enables the drone to move beyond static, one-off interactions toward continuously improving task execution.

Shortcuts

A shortcut is a concise user-defined label that maps directly to a complete multi-step task. The user can at any time request to save a previously executed task, packaging all its steps and progress under an associated label. The user can then request the execution of the saved task using that specific label. The LLM takes the previously executed task into account during new plan generation, ensuring that a behavior that has already proven effective is reused or adapted as needed. In this way, shortcuts provide an intuitive mental handle for complex instructions, allowing users to recall long behaviors through short and memorable labels of their choice.

Shortcuts enhance both efficiency and accuracy in task execution. Efficiency arises because the LLM can ground its reasoning in an existing validated plan rather than generating a full solution from scratch. Since the saved task also preserves the outcomes of previous executions, the LLM can reuse decisions that were already successful, reducing both planning effort and execution latency. This reuse also improves accuracy, as validated behaviors are preserved and unnecessary variations that may emerge when plans are regenerated from scratch are avoided. At the same time, shortcuts reduce cognitive load for the user: instead of recalling long instructions, the user only needs to remember the shortcut label.

Example: *Find apple, then orange, then report to the nearest person.* Consider the enriched instruction: *"Find an apple. Then find an orange and then go to the person closest to you and tell him how many of them you have found"*. After execution, the user may define a shortcut such as *"count apples and oranges"*, which `DACS` associates with the entire task. When later invoked, the shortcut incorporates the validated sequence into the new request, guiding the LLM to reproduce the same plan: detect apples, detect oranges, approach the nearest person, and announce the counts.

Shortcuts support multiple use cases. They can reduce cognitive effort by condensing long requests into short labels, provide fast access to alternative strategies in safety-critical contexts (e.g., quickly choosing between *"plan A"* and *"plan B"*), or preserve privacy by mapping a sensitive task to a neutral label such as *"secret task"*.

Finally, shortcuts also benefit the LLM. The system retains not only the validated plan but also its execution history, including successes, failures, and refinements. By analyzing this record, the LLM can improve its strategies for future tasks. Thus, shortcuts bridge user convenience and system learning: they simplify interactions, ensure predictable results, and enrich the planning knowledge base over time.

Environmental Memory

In `TypeFly`, the system maintains a static view of the environment that is limited to the current scene observed through the drone's FPV camera. Environmental information is processed only at the level of the immediate perception, and once the scene changes, previously observed context is not retained or integrated into future reasoning. As a result, each interaction is effectively isolated, and exploration does not benefit from past observations.

In contrast, `DACS` maintains a representation of the environment that grows over time as

exploration progresses. Each visited region is stored together with its position and the objects perceived from that location. Initially, regions are given generic identifiers (e.g., *region_1*, *region_2*), but the LLM can later rename them once distinctive features are observed, such as recognizing that *region_2* corresponds to a *kitchen*. This process allows the system to gradually build a structured, semantically rich understanding of the environment, starting from scratch and refining it incrementally through interaction. Unlike the static, scene-bound perception in TypeFly, this representation evolves continuously and supports reasoning across multiple exploration steps.

Environmental knowledge can also be injected by the user through the natural language interface. For example, the user may state: *"The environment is composed of two regions, an office at position* $(0, 0)$ *and a corridor at position* $(500, 0)$*, and the office contains these objects ..."*, or upload images to initialize the system's understanding. In this way, the system may start from an approximate but useful model, which is then refined during exploration. Another option is for the user to command the drone to autonomously explore, so that it memorizes environmental information for later tasks. Alternatively, the drone can even remain turned off while the user manually moves it around specific areas, teaching it the features that should be associated with those regions. These multiple modes of interaction allow the user to choose how much control to exercise in shaping the system's knowledge.

Example: *Find an apple.* Suppose the user asks the drone to *"find an apple"* while it is currently located in the bedroom. In `TypeFly`, the system would rely solely on the current FPV scene and initiate a local, blind scan, regardless of any previous exploration. In contrast, if during a prior exploration `DACS` records that apples have been observed in the *kitchen* region, the LLM can immediately direct the drone there instead of scanning the current room. This significantly reduces execution latency by exploiting accumulated environmental knowledge.

Once in the kitchen, the drone verifies whether the apple is still present by performing a new scan. If the apple is no longer there, the memory is updated accordingly by removing outdated entries, after which the system resumes active exploration to search for alternative locations. This example illustrates how environmental memory enables a dynamic understanding of the environment, where exploration progressively enriches the system's knowledge and directly informs future actions, rather than remaining confined to a static snapshot of the current scene.

4.3.5. Visual Environment Awareness

A key innovation introduced in `DACS` is its ability to reason directly over images of the environment. Recent benchmarks such as `CognitiveDrone` highlight the importance of Vision Language Action models reasoning for UAVs, demonstrating how multimodal models can ground decision making in visual context [43]. In `TypeFly`, the LLM relies exclusively on symbolic scene descriptions, namely lists of detected objects produced by the Vision Encoder. In `DACS`, by contrast, raw images captured by the drone can also be transmitted to the LLM. This choice enables the system to exploit the LLM's full visual-spatial reasoning capabilities, allowing it to understand not only which objects are present but also how they are arranged in space, the context in which they appear, and even fine-grained characteristics such as color or personal details like clothing and hair. For example, the same object, such as a bottle, may carry very different meanings depending on whether it is observed in a kitchen or in an office. Unlike the baseline system, which avoids sending images for privacy and latency concerns, `DACS` deliberately prioritizes richer reasoning. Privacy is not considered a constraint, and latency is kept manageable by transmitting images only when necessary, for example to answer a user query such as *"is there a man wearing a red t-shirt in the room?"* or to decide which direction is most promising to explore in order to complete a task. By contrast, simpler queries such as *"is there an apple?"* do not require image transmission, as they can be resolved directly from the YOLO object list produced by the Vision Encoder.

During execution, the drone continuously captures images of its surroundings through the onboard camera. The Vision Encoder still performs YOLO-based detection to extract structured information such as object classes, positions, and bounding boxes. However, this symbolic representation is no longer the only input available: the LLM can also receive the corresponding images, enabling it to combine symbolic detections with direct visual evidence when reasoning about the next steps of a task.

Example: *Find an apple.* Consider the instruction *"find an apple"*. In `TypeFly`, the drone rotates in 45-degree increments and checks the list of detected objects after each step. If no apple is visible after a complete rotation, a new plan is generated through the replanning mechanism, which again relies on the same predefined scanning routine. As a result, the system repeatedly executes equivalent plans without reasoning beyond these symbolic observations.

In `DACS`, the LLM can analyze the captured images directly. After scanning, it may notice contextual cues in one direction, such as a table, a bowl, and a sink that together suggest

a kitchen. Even if no apple is explicitly detected, the LLM infers that the kitchen is the most likely place to find one, so the drone moves toward the kitchen and initiates a new round of inspection. If the apple is still not found, the LLM integrates this new evidence into its reasoning thanks to the Short-Term Memory. If during the kitchen scan the drone observes another nearby region containing other fruits, such as a balcony, the LLM may conclude that this region is now the most promising direction to explore, since areas with fruit often also contain apples. The search therefore continues in a reasoning-driven manner, guided by correlated visual cues rather than by blind exploration.

Through this mechanism, the drone actively selects areas to investigate based on contextual, spatial, and fine-grained visual cues, rather than repeating a fixed scanning procedure. Each decision combines symbolic detections with richer visual context, transforming execution from a rigid sequence of rotations into an adaptive, reasoning-driven search process. In this way, visual environment awareness enables `DACS` to connect perception with reasoning: while `TypeFly` keeps scanning in place by repeatedly regenerating the same predefined scanning procedure, `DACS` continues the search intelligently, exploiting visual, semantic, and contextual clues to navigate toward the most promising regions. This ability to reason about where and how to look next marks an important step toward autonomous, context-aware exploration in human-drone interaction.

4.3.6. User Clarification in Ambiguous Tasks

Another key aspect of `DACS` is its ability to manage ambiguity through proactive dialogue with the user. In `TypeFly`, the system directly translates a natural language instruction into a plan and executes it, assuming that the command is complete and unambiguous. If the instruction lacks sufficient detail, the resulting plan could be incorrect or too generic, leading to inefficient or unintended behavior. To address this limitation, `DACS` introduces a bidirectional communication mechanism that allows the system to request clarifications before generating or executing a plan.

When an ambiguous task is received, the LLM analyzes the user's input and determines whether additional information is needed to generate a meaningful plan. If critical details are missing, such as which object to search for, the system interrupts the planning process and initiates a short dialogue with the user. This interaction is conducted entirely in natural language and aims to resolve uncertainty efficiently without requiring explicit programming or structured input.

Clarification also extends to contextual interpretation. If the drone perceives multiple objects that could satisfy a vague description, for example two people in response to the

task *"follow a person"*, the LLM can ask for disambiguation by saying *"Which person should I follow, the one on the left or the one on the right?"*.

Example: *Find an apple.* If the user simply asks the drone to *"find an object"*, the LLM in `TypeFly` would interpret the request literally and generate a generic plan that scans the environment in search of "object". Such a plan is not useful, since it never resolves the real ambiguity of the command and therefore cannot fulfill the user's actual intention. In contrast, in `DACS`, the LLM recognizes the ambiguity in the term "object". Before proceeding, it requests clarification, for instance *"Which object should I look for?"*. If the user responds *"an apple"*, the LLM incorporates this information into the plan and continues execution as described in previous sections. In this way, a vague command is transformed into a precise instruction that aligns with the user's intent.

Through this proactive questioning mechanism, `DACS` transforms natural language from a one-directional command channel into an interactive dialogue. By ensuring that the user's intent is fully understood before execution, the system reduces the likelihood of failure, minimizes unnecessary exploration, and fosters a more transparent and cooperative form of human-drone interaction.

4.3.7. Flyzones

An additional extension introduced in `DACS` is the concept of *flyzones*: safe operating regions that constrain plan execution. Unlike `TypeFly`, where tasks are executed without explicit spatial boundaries, `DACS` allows users to define flyzones directly in natural language through geometric descriptions of regions and their connections. These bounded regions provide a contextual frame for reasoning and are particularly relevant in both indoor and outdoor scenarios, where limiting the operational space simplifies task execution and aligns the drone's behavior with user intent.

For example, in indoor environments such as warehouses, offices, or industrial facilities, a user may specify: *"The flyzone is composed of two large squares of 7 meters, connected by a small corridor"*. This description is translated into explicit spatial constraints over a plane, which guide plan generation. The LLM incorporates these constraints into its reasoning and ensures that no skill execution places the drone outside the allowed area, preventing it from entering rooms or areas where operation is undesired or unsafe.

Flyzones primarily improve *safety*. By restricting the drone to a well-defined region, they prevent it from entering areas where it could collide with furniture, sensitive objects, or even people, thereby avoiding potential damage or harm. This spatial constraint ensures

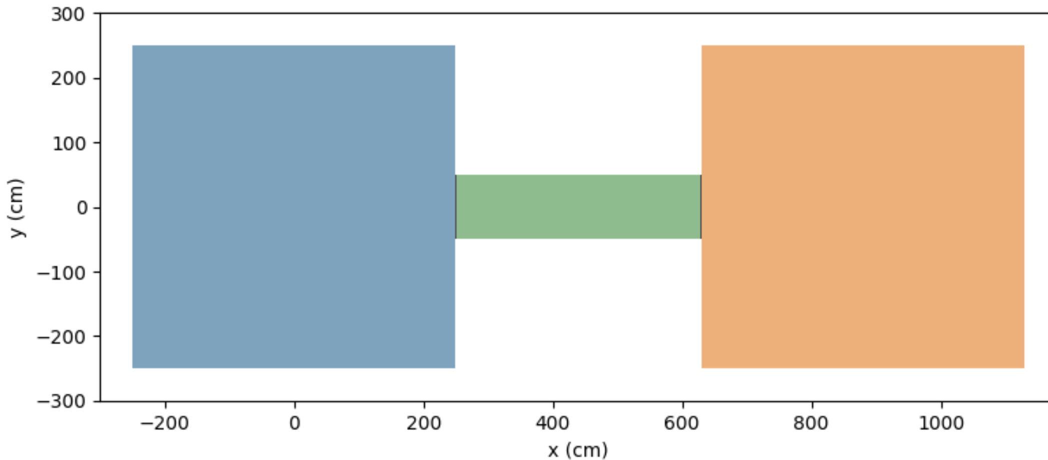


Figure 4.4: Example flyzone composed of five axis-aligned polygons generated by the system. The two large square polygons represent the main rooms (500 cm × 500 cm), connected by a central corridor polygon of dimensions 380 cm × 100 cm. Two thin black vertical polygons located at the corridor entrances denote transition boundaries between the rooms and the corridor. Colors indicate distinct polygonal regions. Together, these polygons define the admissible spatial domain in which the drone is allowed to operate.

that autonomous exploration remains under user control and within safe limits. From a human-drone interaction perspective, flyzones also increase predictability and user trust by making the drone's operating space explicit and controllable.

Beyond safety, flyzones also improve *efficiency* by reducing the action space to only permitted regions, thereby lowering latency, avoiding unnecessary movements, and focusing the search on relevant areas. This is especially valuable in outdoor scenarios such as emergency response or search-and-rescue operations, where users can define geographic boundaries to focus exploration on critical regions affected by a natural disaster. By constraining the search space, the drone can allocate resources more effectively and avoid exploring areas that are irrelevant or inaccessible.

Because flyzones can be created, modified, or removed entirely through natural language, they provide flexibility without requiring technical configuration. They further enhance effectiveness by allowing the user to define regions where the drone is most likely to complete the task successfully, both in bounded indoor environments and in large-scale outdoor missions.

Example: *Find an Apple.* In TypeFly, the instruction *"find an apple"* triggers a scanning routine without movement. In contrast, DACS enables the drone to actively explore, as described in Section 4.3.5. For this reason, it is important to constrain the drone's movement domain, particularly in indoor environments.

With DACS, the user can define a flyzone such as *"a square of 5 meters representing the kitchen, connected by a corridor of 3.8 meters to another square representing the dining room"*. This configuration is illustrated in Figure 4.4. The drone can then navigate between the two squares through the specified corridor while remaining within the allowed operating area. When the flyzone representation accurately reflects the real environment, the system leverages these constraints to generate effective plans that fulfill the task while avoiding unnecessary exploration outside the regions of interest.

4.3.8. Integrated Example

This narrative illustrates how DACS transforms a vague command into a refined, reusable, and personalized behavior. Through clarification, visual awareness, environmental memory, flyzones, high-level skill creation, feedback integration, and shortcuts, the system not only fulfills the immediate task but also continuously evolves across sessions, aligning more closely with the user's expectations over time.

To demonstrate this, consider the following multi-session scenario.

Session 1: Initial request. The user begins with a vague instruction: *"Find a fruit"*. The LLM immediately detects ambiguity and asks for clarification (Section 4.3.6): *"Which fruit should I look for?"* The user responds: *"An apple"*.

At this point, the drone consults its environmental memory (Section 4.3.4), but since no apples have been recorded yet, it must actively explore. Movements are constrained within the previously defined flyzone, which includes the kitchen, dining room, and the corridor connecting them (Section 4.3.7). After scanning, the drone transmits captured images to the LLM. Although YOLO does not initially detect an apple, the LLM reasons from visual context (Section 4.3.5): it notices a fruit bowl on the kitchen table and directs the drone closer. The apple is then successfully detected behind the bowl, and the system reports back to the user.

The LLM encapsulates this sequence of actions into a new high-level skill `search_object(fruit_name)` (Section 4.3.3). From now on, similar tasks can be executed efficiently without regenerating a plan from scratch (e.g., `search_object("orange")`).

Session 2: User feedback. During the next interaction, the user again requests: *"Find an apple"*. This time, the LLM directly applies the new `search_object("apple")` skill. The apple is quickly located in the kitchen thanks to environmental memory (Section 4.3.4). After execution, the user provides feedback (Section 4.3.4): *"Next time, also tell me how many apples you see"*.

The system saves this preference in memory and will automatically apply it to similar tasks in the future. The updated version will then detect and count all apple instances, and report the total back to the user.

Session 3: Shortcut. Later, the user issues a more complex instruction: *"Find apples, then find oranges, and tell me the counts of both"*. The drone executes this task by composing calls to `search_object("apple")` and `search_object("orange")`. Since the execution is successful, the user defines a shortcut (Section 4.3.4): *"Save this as `fruits count`"*. From this point onward, invoking the shortcut *"fruits count"* will not simply replay the same sequence; instead, the stored plan and its execution history are provided to the LLM as additional context. This ensures that the reasoning process can reuse and adapt the validated strategy, while still remaining flexible to changes in the environment.

5 | Implementation

In this chapter we describe the implementation of DACS, the adaptive and context-aware HDI system introduced in Chapter 4. We detail how the architectural principles and interaction mechanisms defined at design time are instantiated in a concrete, executable prototype.

The objective of this implementation is not to deliver a production-ready or optimized system, but to provide a research-oriented platform that enables experimentation and validation of the proposed approach under realistic operating conditions. Accordingly, the implementation prioritizes modularity, observability, and ease of modification over performance optimization or deployment robustness. This design choice ensures that individual components can be independently inspected, refined, and extended throughout the research process.

DACS realizes a runtime pipeline that supports the interpretation, execution, and evaluation of natural-language tasks within a closed control loop. A user provides a task expressed in natural language through a chat-based interface. This input is processed by a set of specialized LLM instances, each associated with a well-defined reasoning role, as described in Section 4.3.2.

The remainder of this chapter is organized as follows. Section 5.1 revisits the baseline TypeFly implementation, which serves as the architectural and execution foundation upon which DACS is built. This section summarizes the key runtime components and execution model inherited from the baseline system. Section 5.2 then introduces the layered software architecture of DACS, describing how hardware abstraction, skills, execution, and reasoning are organized within a unified edge-based runtime. Section 5.3 describes the implementation of the adaptation and awareness mechanisms introduced in Chapter 4. It focuses on the internal representations used by the system, including the context graph and flyzone, and outlines how these representations support adaptive planning and environment-aware behavior. Finally, Section 5.4 provides a detailed discussion of the individual LLM instances that realize the distributed reasoning framework of DACS.

5.1. Baseline TypeFly Implementation

Before detailing the implementation of `DACS`, it is necessary to describe the implementation of the baseline `TypeFly` system introduced in Chapter 3, as it constitutes the architectural and software foundation upon which the proposed extensions are built. In this section we summarize the concrete implementation choices of `TypeFly`, focusing on its runtime components, execution model, and system-level constraints, as originally presented by Chen et al. [19].

`TypeFly` is implemented as an end-to-end system that enables a quadrotor drone to execute tasks expressed in natural language by combining a cloud-hosted LLM, edge-based perception, and a domain-specific language. The system follows a distributed architecture composed of three main elements: a DJI Tello drone equipped with an onboard RGB camera and wireless communication, an edge server responsible for perception and execution, and a remote LLM service accessed via a network API.

The drone is designed to be computationally lightweight and is responsible exclusively for image acquisition and low-level actuation. It streams camera frames to the edge server and executes motion commands received in return. All high-level reasoning, perception, and task execution logic are performed offboard, simplifying onboard software and allowing the use of computationally intensive models without affecting flight stability or responsiveness.

On the edge server, `TypeFly` integrates three core software components. The first is the `Vision Encoder`, implemented using YOLOv8 [36] real-time object detection models, which processes incoming camera frames and produces a symbolic scene description. Rather than transmitting raw images to the LLM, `TypeFly` converts visual input into compact textual representations, reducing bandwidth usage, LLM inference cost, and privacy exposure.

The second component is the `LLM Controller`, which orchestrates interaction with the remote LLM, OpenAI GPT-4 [2]. For each user task, the controller constructs a planning prompt by combining the user's task description, the current scene description generated by the `Vision Encoder`, and a catalog of available robot skills. The prompt is designed to encourage the generation of concise and syntactically valid execution plans. The LLM responds with a task plan expressed in `MiniSpec`, a custom domain-specific language, as described in Section 3.3.1.

The third component is the `MiniSpec Interpreter`, which validates and executes the plan generated by the LLM. Before execution, the interpreter performs syntactic validation and ensures that only supported skills are invoked. During execution, each MiniSpec

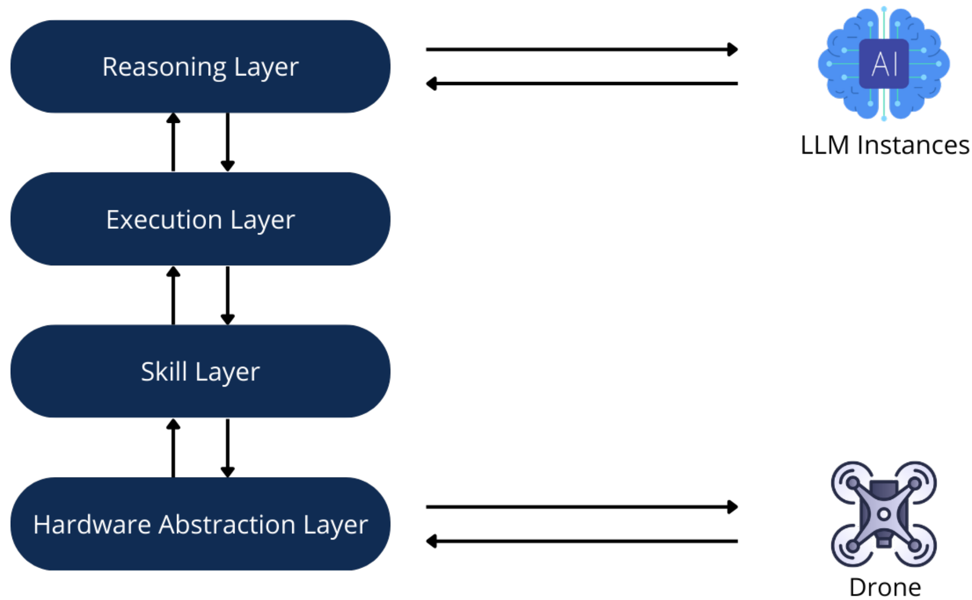


Figure 5.1: Layered software architecture of `DACS`.

instruction is translated into a corresponding Python function call that interfaces with perception or robot control. In the reference `TypeFly` implementation, high-level skills are statically defined and limited to two perception-oriented primitives: `scan`, which performs a rotational search for a specified object, and `scan_abstract`, which searches based on an abstract natural-language description of an object.

5.2. Software Architecture

Building upon the baseline `TypeFly` implementation introduced in Section 5.1, `DACS` adopts a layered software architecture that integrates aerial robotics, context-aware reasoning, and adaptive interaction within a unified runtime framework. All runtime components execute on the edge server, while the aerial platform remains responsible exclusively for sensing and actuation. LLM-based reasoning is performed through cloud-hosted models accessed via remote API calls, allowing computationally intensive inference to be offloaded from both the edge infrastructure and the drone itself. Although physically external to the edge environment, these LLM services are logically integrated into the system through the `Reasoning Layer`, which orchestrates their invocation, constructs prompts, and mediates the flow of contextual information between reasoning, execution, and memory components.

The architecture is organized into four layers of increasing abstraction, illustrated in Figure 5.1. This layered design promotes modularity and a clear separation of responsibilities

between hardware interfacing, executable behaviors, program interpretation, and the orchestration of external LLM-based reasoning services.

5.2.1. Hardware Abstraction Layer

The `Hardware Abstraction Layer` provides a uniform interface between the software stack and the physical robotic platform, insulating higher layers from platform-specific communication protocols, sensing modalities, and actuation constraints. All interaction with the drone and the external localization system is mediated through the `RobotWrapper` abstraction, which exposes high-level capabilities such as motion execution, video streaming, and pose retrieval while hiding low-level control details.

Multiple concrete implementations of the `RobotWrapper` interface are provided. The `TelloWrapper`, responsible for DJI Tello flight control, and the `VirtualRobotWrapper`, used for simulated execution during development and testing, are inherited from the original `TypeFly` implementation. In contrast, `DACS` introduces the `CrazyflieWrapper`, a dedicated wrapper added to interface with the external Lighthouse localization system [9] and provide continuous drone pose estimation to the higher architectural layers.

5.2.2. Skill Layer

The `Skill Layer` defines the executable behaviors available to the system and bridges structured planning with physical execution. Skills represent the action vocabulary used by MiniSpec programs, enabling the system to translate high-level intent into concrete operations without exposing low-level control details to external reasoning services.

Low-level skills correspond to atomic operations directly connected to hardware control, perception queries, memory management, or system utilities. Most low-level skills are inherited from the original `TypeFly` implementation and preserve the same execution semantics. In `DACS`, this baseline is extended with additional low-level skills designed to interface with new reasoning capabilities. These skills act as explicit invocation points for cloud-hosted LLM reasoning services: each skill encapsulates the construction of a structured prompt, the execution of an LLM call, and the transformation of the returned output into a format usable by the execution pipeline. The detailed design of these LLM-coupled skills and the associated reasoning modules is presented in Section 5.2.5.

High-level skills encapsulate reusable behaviors defined as MiniSpec programs composed of multiple low-level skills. During execution, a high-level skill is expanded inline and interpreted by the `Execution Layer`. The baseline library of high-level skills is inherited

from TypeFly; however, DACS extends this model by allowing new high-level skills to be created dynamically at runtime.

5.2.3. Execution Layer

The Execution Layer is responsible for interpreting and executing MiniSpec programs obtained through the reasoning pipeline. It acts as the runtime bridge between structured plans and concrete skill invocations, managing control flow, execution state, and replanning signals while remaining independent from both low-level hardware control and external reasoning services.

At the core of this layer lies the MiniSpecInterpreter, which implements a streaming parser and executor for MiniSpec programs. The interpreter architecture is inherited from the original TypeFly system, preserving compatibility with existing MiniSpec programs. While the baseline interpreter executes MiniSpec instructions directly as they are parsed, DACS introduces targeted extensions to distinguish between executable MiniSpec code and MiniSpec fragments that must be treated as data rather than immediately executed. This distinction is required to support skills such as add_skill, in which a high-level skill definition is passed as an argument. In this case, the MiniSpec definition string must be preserved, validated, and registered for future use, rather than being parsed and executed at invocation time.

5.2.4. Reasoning Layer

The Reasoning Layer constitutes the highest level of abstraction on the edge server and is responsible for orchestrating all interactions with cloud-hosted LLM reasoning services. Rather than performing reasoning internally, this layer manages prompt construction, contextual grounding, and the structured exchange of information between external reasoning processes and the rest of the runtime architecture.

The overall orchestration mechanism is inherited from the original TypeFly system through the LLMController component. In the baseline implementation, the controller mediates interaction with a single planning-oriented LLM. In DACS, this design is extended to support multiple specialized LLM calls, together with additional mechanisms for memory management, context-aware reasoning, and adaptive interaction workflows.

The LLMController maintains the current task state, environment representations, available skills, execution history, and persistent user-specific preferences. Acting as a coordination hub, it aggregates user interaction, execution feedback, and contextual information

before invoking the appropriate external reasoning services.

5.2.5. LLM Instance Abstraction

Within the layered software architecture, the `Reasoning Layer` interfaces with a set of cloud-hosted LLM-based reasoning services that perform specialized cognitive functions during execution. In `DACS`, these external reasoning services operationalize the distributed reasoning model introduced in Chapter 4. Each service corresponds to a specialized reasoning role and is activated on demand through dedicated low-level skills inserted into the execution plan by the central reasoning component, as illustrated in Figure 5.2.

Definition of an LLM instance. In `DACS`, we formalize each external LLM-based reasoning service as an *LLM instance*, namely a concrete and independent reasoning component characterized by:

- a *fixed system prompt* that specifies the role, scope, and constraints of the instance;
- a *dynamically generated user prompt* that injects task-specific and runtime dependent information;
- a set of *configuration parameters*, including model choice, reasoning effort, and verbosity.

Each instance is designed to perform a narrowly scoped reasoning function, such as plan generation, perceptual querying, environment modeling, or feedback processing. This explicit role separation reduces prompt complexity, improves output predictability, and enables independent tuning of reasoning behavior.

Deployment model and statelessness. All LLM instances are deployed as remote services on the OpenAI platform and are accessed programmatically through the `OpenAI Responses API` [47]. Each invocation corresponds to a stateless API request in which the system prompt, the runtime-generated user prompt, and the instance-specific configuration parameters are explicitly provided. As a result, LLM instances do not retain internal state across calls. All persistent knowledge, including execution history, user feedback, learned skills, and environmental representations, is maintained explicitly by the `Reasoning Layer` and injected into user prompts when required.

In the original `TypeFly` system, only two LLM instances are used: a *planning instance* responsible for generating MiniSpec plans, and a *querying instance* used for limited runtime reasoning. In contrast, `DACS` implements nine distinct LLM instances (Figure 5.2), each

dedicated to a specific adaptive or awareness-related capability, as detailed in Section 5.4. All instances rely on the same underlying LLM API but differ in their system prompts, user prompts, and runtime configuration.

Low-Level skills and Plan LLM orchestration. A central implementation principle of `DACS` is that LLM instances are never invoked directly by the controller. Instead, access to each instance is encapsulated through dedicated low-level skills, which act as explicit invocation interfaces.

`Plan LLM` is the central reasoning component of the system and the only instance capable of generating MiniSpec execution plans. All other LLM instances are auxiliary and can be activated exclusively through low-level skill calls explicitly emitted by `Plan LLM`. Each skill invocation triggers the corresponding LLM instance, constructs the appropriate prompt, performs the API call, and parses the returned output into a form usable by the controller or `Plan LLM`.

For example, as illustrated in Figure 5.2, when `Plan LLM` infers that the user task cannot be fulfilled within the current region based on available perception and execution outcomes, it emits an `explore_direction` skill call, triggering the `Choose Direction LLM` instance, as we will see in Section 5.4.7.

`Short-Term Memory LLM` differs from all other instances. Unlike auxiliary instances, it is not associated with any low-level skill and cannot be invoked during plan execution. It is triggered exclusively at the end of each execution iteration, and only when the task has not been completed and a new planning iteration is required.

5.3. Environmental Internal Representation

To support adaptive planning, exploration, and memory, `DACS` maintains an explicit internal representation of the environment, referred to as the *context graph*. This graph captures what the system has observed so far about the environment and provides a shared, structured model consumed by planning, querying, exploration, and memory-related components.

The conceptual role of the context graph was introduced in Chapter 4. In this section, we focus exclusively on its concrete implementation, runtime evolution, and internal representations.

Graph structure and runtime management. From an implementation standpoint, the context graph is represented as a structured JSON object that models both spatial semantics and runtime state. The graph encodes *regions*, which denote semantically meaningful areas of the environment and are associated with explicit two-dimensional coordinates, as well as *objects*, corresponding to entities detected through a YOLO-based perception pipeline. Relationships within the environment are captured through *object_connections*, which associate objects with the regions they belong to, and *region_connections*, which describe adjacency or navigational connectivity between regions. In addition, the graph maintains a *current_position* field representing the current pose of the drone together with its associated region.

The lifecycle of the context graph is managed by a dedicated software component, the `GraphHandler`, which is responsible for loading the graph from persistent storage, applying incremental updates at runtime, validating structural consistency, and serializing the updated representation back to disk as a JSON file. All other system components access the graph exclusively through the `GraphHandler`, ensuring a single authoritative representation of the environmental state.

Runtime region creation. Regions are not defined a priori, but are created dynamically at runtime as the drone explores the environment. A new region is instantiated whenever the drone moves farther than a configurable distance threshold from all existing region centroids. New regions are initially assigned automatically generated identifiers of the form `region_0`, `region_1`, `region_2`, and so on, using an incremental integer counter. At runtime, regions may later be assigned human-interpretable semantic names based on visual context, as discussed in Section 5.4.7.

Whenever the drone transitions from one region to another, the `GraphHandler` component records this event by adding a corresponding entry to *region_connections*. These connections are created incrementally at runtime and encode only the navigation paths that have been *empirically executed* by the drone during exploration. As such, *region_connections* provide the LLM with a record of which region-to-region transitions have already been attempted and successfully traversed.

Importantly, the absence of a connection between two regions does *not* imply that the transition is impossible in the real environment. A missing edge simply indicates that the drone has never attempted to move between those regions. If the flyzone representation permits motion between the corresponding coordinates, then the transition is geometrically feasible even if it is not yet represented in the context graph. In this sense, the context graph captures the history of exploration rather than the full set of reachable

paths, while the flyzone remains the authoritative source of global navigation constraints and spatial feasibility.

Drone pose and region assignment. In addition to regions and objects, the context graph explicitly stores the current pose of the drone. The pose is represented by four values: three-dimensional position (x, y, z) and yaw orientation, and is stored together with the identifier of the region in which the drone is currently located.

Region membership is not reported directly by the localization system but is instead derived by the `GraphHandler` component. Given the current drone position and the coordinates of all known regions, the `GraphHandler` assigns the drone to the closest region whose spatial extent contains the current position. This association is recomputed whenever the drone pose or the region set is updated, ensuring that the context graph always reflects a consistent mapping between physical position and semantic region.

Sparse and dense graph representations. The context graph is maintained in two complementary representations: a *sparse*, structured JSON format and a *dense*, linearized textual format. Both representations encode the same environmental state but are optimized for different purposes within the system.

The sparse JSON representation is used for persistence, structured validation, and internal manipulation by system components. For example, in an office-like environment perceived at runtime, the sparse representation may take the following form:


```
{
  "objects": [
    { "name": "person" },
    { "name": "tv" },
    { "name": "remote" },
    { "name": "chair" },
    { "name": "laptop" }
  ],
  "regions": [
    {
      "name": "office",
      "coords": [0.0, 0.0]
    }
  ],
  "object_connections": [
```

```
    ["office", "person"],
    ["office", "tv"],
    ["office", "remote"],
    ["office", "chair"],
    ["office", "laptop"]
  ],
  "region_connections": [],
  "current_position": {
    "coords": [0.0, 0.0, 0.0, 0.0],
    "region": "office"
  }
}
```


The dense representation provides a compact, human-readable summary of the same context graph and is primarily used when injecting environmental context into LLM prompts, where token efficiency and readability are critical. In this format, information is linearized and aggregated around regions. For the same example, the dense representation is:

```
office(0.0,0.0): person, tv, remote, chair, laptop
| POS: [0.0, 0.0, 0.0, 0.0] @ office
```

The `GraphHandler` component maintains consistency between the sparse and dense representations and generates the dense form on demand.

Flyzone representation. The flyzone is managed by a dedicated software component, the `FlyzoneHandler`, which is responsible for storing, validating, and updating the geometric representation. Internally, geometric operations are performed using `Shapely` [29], a Python computational geometry library that provides robust planar geometry representations and supports operations such as containment tests, intersection, union, and validity checking.

The flyzone is stored as a structured JSON object encoding a list of closed polygons, where each polygon is defined by an ordered sequence of (x, y) coordinates expressed in centimeters. For example:


```
"points_list": [
  [[50,0], [49,9], ..., [50,0]],
  [[-25,45], [25,45], [25,125], [-25,125], [-25,45]],
  [[30,150], [29,155], ..., [30,150]]
```

]

5.4. LLM Instances

This section presents the individual LLM instances that compose the distributed reasoning framework of `DACS`. Building upon the abstraction introduced in Section 5.2.5, we first describe the shared configuration principles adopted across all instances, and then provide a detailed discussion of the role, inputs, and behavior of each specialized module. Together, these components realize the adaptive planning, memory management, environment modeling, and exploration capabilities integrated into the software architecture.

5.4.1. Instances Configuration

Each LLM instance in `DACS` is configured through a common set of parameters exposed by the OpenAI Responses API, namely the underlying model, verbosity, and reasoning effort. *Verbosity* controls the amount of natural-language content produced in the final response and can be configured at four discrete levels: `low`, `medium`, or `high`. *Reasoning effort* controls the amount of internal computational processing the model allocates before producing an answer and can be set to `minimal`, `low`, `medium`, or `high`. All instances rely on the same underlying model, `GPT-5`, which we adopt consistently across the implementation as the most advanced model used during development and experimental testing.

We determine the specific configuration of verbosity and reasoning effort for each instance through an iterative process of experimentation and fine-tuning, with the objective of balancing reasoning quality, responsiveness, and overall system usability. *Verbosity* is kept low across all LLM instances. This choice is motivated by two factors. First, since `DACS` builds upon `TypeFly`, it adopts `MiniSpec` as a compact execution language based on minimal syntactic overhead. This design allows semantically rich and structurally complete execution plans to be expressed with a very small number of tokens. As a result, even complex behaviors can be represented concisely, enabling the system to generate exhaustive and fully specified plans while maintaining low verbosity and reduced token usage. Second, even when an instance produces natural-language output, prompts explicitly request short, structured answers rather than free-form explanations. As a result, the system prioritizes targeted information extraction over open-ended narrative responses.

Reasoning effort is configured to be either minimal or low, depending on the role of the LLM instance in the execution pipeline. Increasing reasoning effort generally improves

LLM instance	Verbosity	Reasoning effort
Plan	`low`	`low`
Query	`low`	`minimal`
Short-Term Memory	`low`	`minimal`
Choose Direction	`low`	`low`
Create Graph	`low`	`low`
Create Flyzone	`low`	`low`
Save Task Feedback	`low`	`minimal`
Update Universal Feedback	`low`	`minimal`
Retrieve Task Feedback	`low`	`minimal`

Table 5.1: Summary of verbosity and reasoning effort configuration for the LLM instances in `DACS`.

output consistency and robustness, but directly increases response latency. For instances involved in runtime decisions that directly affect execution, such as querying the environment, we generally adopt minimal reasoning effort to preserve system responsiveness. For instances that perform structured transformations or contextual analysis, such as context graph construction or flyzone generation, we adopt low reasoning effort to achieve a favorable trade-off between output quality and response time. Table 5.1 summarizes the specific verbosity and reasoning-effort configuration adopted by each LLM instance, anticipating the detailed descriptions provided in the following sections.

5.4.2. Plan LLM Instance

The `Plan LLM` instance is the central reasoning component of `DACS` and the only instance responsible for generating executable MiniSpec plans. It represents the single point at which task interpretation, plan generation, and coordination of auxiliary reasoning capabilities are consolidated.

The configuration of this LLM instance uses *low reasoning effort*. Although reducing reasoning effort would further decrease planning latency, the `Plan LLM` is responsible for the most demanding reasoning task in the system. It must jointly interpret user intent, current environmental knowledge, available skills, execution history, and persistent user preferences to generate safe, syntactically valid, and semantically coherent MiniSpec plans.

We govern the behavior of the Plan LLM instance through a fixed system prompt that defines its operational rules, constraints, and expected outputs. The system prompt encodes: (i) the role of the instance as a MiniSpec plan generator, (ii) strict syntactic and semantic constraints of the MiniSpec language, (iii) safety rules such as mandatory flyzone enforcement, (iv) replanning and termination conditions, and (v) rules for interpreting and

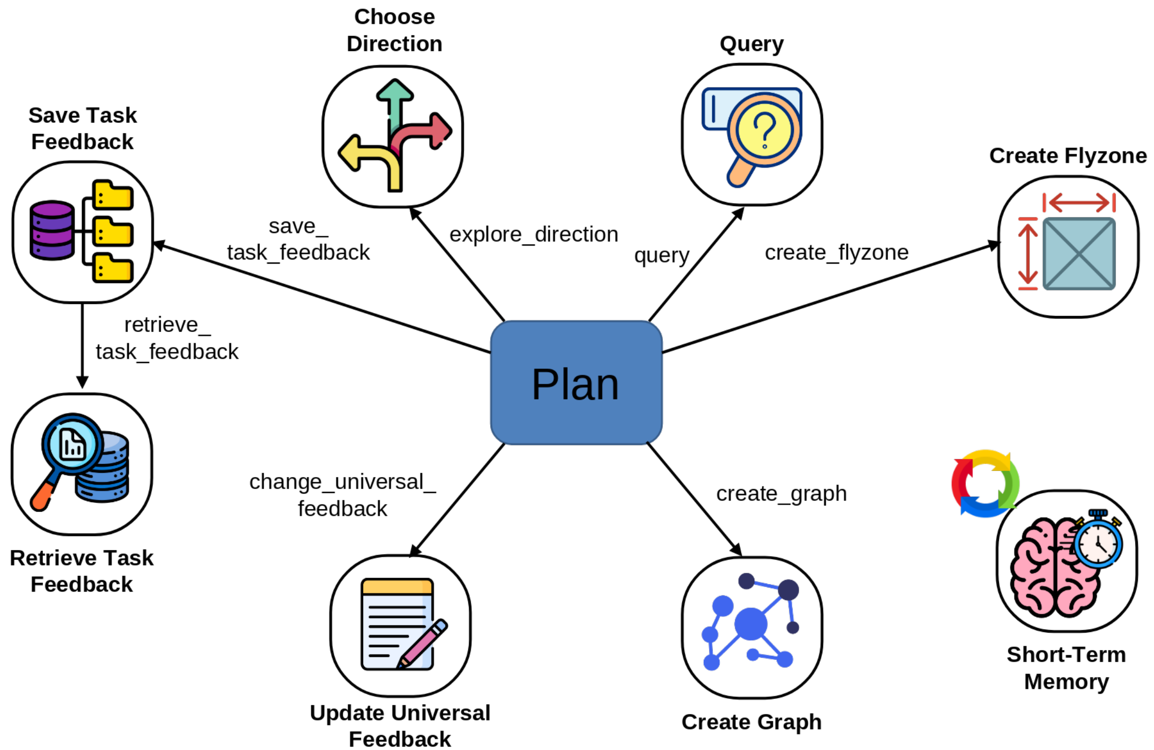


Figure 5.2: Overview of the multi-LLM interaction pattern in `DACS`. The `Plan LLM` acts as the central orchestrator and triggers auxiliary LLM instances through dedicated low-level skills during plan execution. All auxiliary instances are invoked on demand, except for the `Short-Term Memory` instance, which is activated only when a task requires a new planning iteration based on the outcomes of previous executions.

handling user feedback and preferences.

A critical component of the system prompt is the inclusion of a curated set of execution examples. Rather than serving as templates to be replicated, these examples define a behavioral reference frame that guides how the `Plan LLM` translates reasoning into valid MiniSpec programs. The examples are intentionally heterogeneous and cover multiple planning situations encountered at runtime. Some illustrate how recurring reasoning patterns can be encapsulated into reusable high-level skills, enabling the system to abstract common behaviors such as scan-explore cycles. Others demonstrate how the context graph should influence navigation decisions, encouraging the model to exploit previously discovered regions instead of performing unnecessary exploration. Additional examples encode safety-oriented behaviors, such as decomposing trajectories into intermediate waypoints that respect flyzone constraints, or handling environment descriptions by invoking graph- or flyzone-creation routines without triggering physical execution. Finally, dedicated examples clarify the distinction between universal user preferences and task-related

feedback, ensuring that persistent memory updates follow consistent semantic rules.

A representative example included in the curated execution set illustrates how the `Plan LLM` exploits the *context graph* to generate navigation plans grounded in previously acquired environmental knowledge. In this scenario, the user asks the robot to return to a previously visited region. The context graph indicates that the robot is currently located in `region_1` at coordinates $(0, 150)$, while `region_0` corresponds to the home position at $(0, 0)$. Since the flyzone allows direct motion between these coordinates, the planner can generate a feasible trajectory using the `go_position` skill. Rather than issuing a single low-level command, the example demonstrates how recurring navigation patterns can be abstracted into a reusable high-level skill.

```
Task: Come back to region_0, because I need you here

Graph state:
- Current region: region_1
- Current position: [0, 150]
- Target region: region_0 at [0, 0]

Planning rationale:
The goal is to return to a known region whose coordinates are stored in
the context graph. The trajectory lies inside the flyzone, therefore a
direct motion using gp is feasible. To encourage reusable planning
patterns, a new high-level skill return_home is created.

Generated MiniSpec plan:
add_skill("return_home",
   "Return to the home region (0,0) at height 150",
   "go_position(0, 0, 150); log_user('Returned to region_0 at [0, 0].')");
return_home();
```

This example conveys several behavioral patterns that the curated prompt examples aim to reinforce. First, planning decisions are grounded in the structured environmental memory provided by the context graph rather than derived solely from immediate perception. Second, the planner is encouraged to encapsulate frequently occurring navigation behaviors into reusable high-level skills, improving modularity and readability of generated MiniSpec programs. Finally, motion commands are produced only when they respect flyzone constraints, ensuring that generated plans remain consistent with the geometric safety model enforced by the execution pipeline.

High-level skills and user-specific skill sets. Unlike the baseline `TypeFly` system, where high-level skills are statically defined at initialization, `DACS` allows new high-level skills to be created dynamically at runtime. During planning, `Plan LLM` can invoke the `add_skill` low-level instruction to define a new high-level skill by specifying its name, description, and a MiniSpec definition composed of existing skills.

Predefined low- and high-level skills are stored in a shared configuration file, while user-defined high-level skills are maintained separately for each user. Users are identified by a unique username, and the active user context can be changed through the `set_username` low-level skill. This enables personalized and persistent skill libraries to evolve over time without interfering across users.

5.4.3. Query LLM Instance

The `Query LLM` instance is responsible for answering localized questions about the current environment state during plan execution. As shown in Figure 5.2, this instance is invoked through the low level skill `query` whenever the `Plan LLM` requires additional perceptual confirmation before continuing execution.

The ability to query the environment is already present in `TypeFly`. In that setting, the `Query LLM` operates exclusively on symbolic perception output. The controller provides a list of objects detected by the YOLO based `Vision Encoder`, together with a natural language question generated by the `Plan LLM`. Based on this input, the `Query LLM` returns a concise answer, typically indicating the presence or absence of a target object or confirming a simple condition. As a result, reasoning is constrained to what is explicitly encoded in the symbolic object list.

In `DACS`, we extend this capability by enabling multimodal querying. In addition to the symbolic object list, the `Query LLM` also receives a raw camera image of the current scene. The system prompt constrains the `Query LLM` to a narrow scene level question answering role and enforces a strict input/output structure. Inputs consist of a natural language query produced by the `Plan LLM`, the list of currently detected objects with their attributes, and the raw camera image. Outputs are restricted to a single value whose format depends on the query type, namely a boolean value for yes no questions, an exact integer for counting queries, or a specific object identifier for selection queries. A representative example of this interaction is shown in Figure 5.3.

Given the highly focused nature of these queries, such as counting visible objects, verifying the presence of an object described by simple attributes, or selecting among a small set of candidates, the `Query LLM` is configured with *minimal reasoning effort*. Empirically,

Figure 5.3: Example input output interaction of the `Query LLM` instance in `DACS`. The input includes the current camera image, the list of objects detected by the YOLO based Vision Encoder with their bounding boxes, and a natural language query such as *"How many apples are there in the scene"*. By jointly reasoning over the symbolic detections and the visual content of the image, the `Query LLM` instance produces a concise answer, in this case an integer count of the apples visible in the scene.

we observe that increasing reasoning effort does not improve answer quality for these narrowly scoped perceptual tasks, while it directly increases response latency.

5.4.4. Short-Term Memory LLM Instance

The `Short-Term Memory LLM` instance summarizes the outcome of each execution cycle and produces a concise natural-language description of what has just occurred. An *execution cycle* consists of a MiniSpec plan generated by the `Plan LLM`, its physical execution by the drone, and the resulting observations. If the plan does not fulfill the user request, a new cycle is triggered through the `replan` skill, placed at the end of the plan; reaching this instruction indicates that the current plan has terminated without achieving the task objective.

The system prompt constrains this instance to a well-defined summarization task and restricts the output to a single JSON field, `iteration_summary`, enforcing consistency and making the result directly consumable by the controller. The prompt explicitly

instructs the instance to focus on the outcomes of low-level skill calls, to connect the current iteration with previous ones when relevant, and to highlight progress, failures, or replanning conditions.

Consider the following MiniSpec plan executed during an iteration:

```
?so(["book"])!=False->Truerp();
```

In this example, the high-level skill `so` (`search_object`) is invoked to locate a `"book"` in the current region. The conditional expression checks whether the search returned a non-`False` value, indicating success. If the condition fails (i.e., the object is not detected), execution proceeds to `rp` (`replan`), which triggers replanning. At runtime, the execution trace records that the search returned `False` and that replanning was initiated.

Given this structured trace, the `Short-Term Memory LLM` produces a semantic summary such as:

```
"The robot attempted to find a book in the current room using the
search routine, but the object was not detected.  An exploration step
was triggered and replanning was initiated to continue the task."
```

This transformation converts a low-level symbolic execution trace into a compact natural-language description that captures both the outcome and its implications for subsequent planning.

The configuration of this LLM instance uses *minimal reasoning effort*, as the required task is limited to structured summarization and does not involve complex inference or decision making.

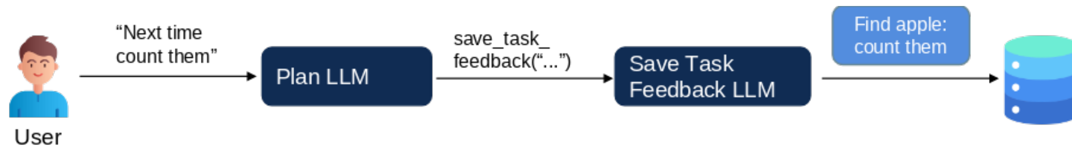
5.4.5. Save Task Feedback LLM Instance

The `Save Task Feedback LLM` instance implements a core component of the long-term memory described in Section 4.3.4. Its purpose is to persist task-related user feedback by consolidating the outcome of a completed execution together with the corresponding user input into a structured semantic representation. This representation can later be retrieved to influence the planning of semantically similar tasks.

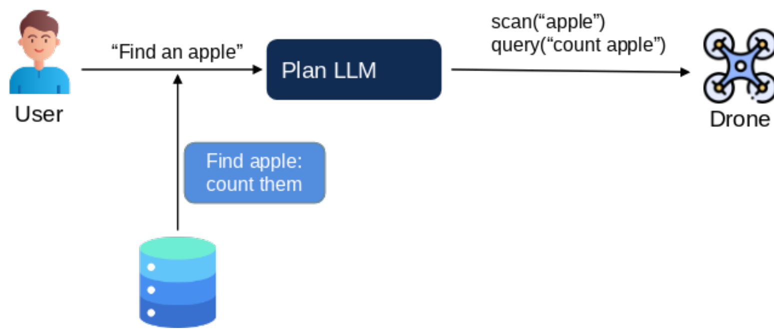
Figure 5.4 illustrates the runtime interaction supporting this mechanism. A user first issues a request that requires physical execution by the drone, for example *"find an apple"*. The request is processed by the `Plan LLM`, which generates an execution plan invoking the appropriate skills. The drone executes the plan and interacts with the environment.

After execution terminates, the user may provide additional messages through the chat

(a) Task execution through Plan LLM.



(b) User provides task-related feedback, which is stored in long-term memory.



(c) Feedback retrieved and applied to new plan.

Figure 5.4: Sequence diagram for task-related feedback persistence and reuse. (a) The user issues a task and the `Plan LLM` generates an execution plan that is carried out by the drone. (b) After execution, the user provides task-related feedback, which triggers the `Save Task Feedback LLM` and results in a structured summary stored as long-term memory. (c) During a subsequent task, previously stored feedback is retrieved and injected into the planning context, allowing the `Plan LLM` to generate an adapted MiniSpec plan that reflects prior user preferences.

interface, as illustrated in Figure 5.4 (b). When a new message arrives, the `Plan LLM` first determines whether it represents a request for a new physical action or feedback referring to the most recent execution. If the message is interpreted as task-related feedback, the `Plan LLM` generates a MiniSpec plan that includes the low-level skill `save_task_feedback`, passing the user feedback text as an argument.

When the `save_task_feedback` skill is executed, the controller constructs the user prompt for the `Save Task Feedback LLM` by combining three elements: the original task description (*"find an apple"*), the execution outcome summarized by the `Short-Term Memory LLM` (Section 5.4.4), and the user feedback text (*"next time count them"*). The LLM produces a compact structured summary that captures the task objective, observed outcome, and the user feedback as a set of *lessons learned*. Since this process involves primarily

summarization and abstraction rather than complex reasoning, the instance is configured with *minimal reasoning effort* in order to reduce latency within the interactive replanning loop.

The generated summaries are stored in a vector database that implements the long-term memory for task-related feedback. For each stored entry, a dense embedding is computed exclusively from the original task description using the OpenAI `text-embedding-3-small` model. During subsequent planning phases, semantic retrieval is performed before each new planning invocation. When a new task request arrives, like in Figure 5.4 (c), the system computes an embedding of the task description and queries the user-specific vector store for entries whose embedding distance falls below a configurable threshold. The retrieved summaries are then injected into the user prompt provided to the `Plan LLM`.

Importantly, retrieval does not enforce the applicability of a stored lesson; instead, the `Plan LLM` evaluates the semantic relevance of each retrieved entry and decides whether it should influence the generated plan. For instance, feedback collected after a task such as *"find an apple"* may include both general preferences (e.g., *"when finding a fruit, do . . . "*) and task-specific instructions (e.g., *"when finding an apple, do . . . "*). If the user later requests *"find a banana"*, both entries may be retrieved due to semantic similarity. However, during planning the `Plan LLM` applies only the fruit-level preference while ignoring apple-specific constraints, demonstrating selective reuse of long-term knowledge.

Deletion of task-related feedback. The system also supports the removal of previously stored task-related feedback through a mechanism that reuses the same symbolic planning framework adopted for feedback persistence. In this case, `Plan LLM` inserts the low-level skill `delete_task_feedback` into the generated MiniSpec plan.

When the `delete_task_feedback` skill is executed, the controller invokes the `retrieve_task_feedback` procedure. As in the retrieval step illustrated in stage (c) of Figure 5.4, the system computes an embedding from the task description referenced by the deletion request and queries the user-specific vector database to obtain candidate feedback entries whose task embeddings fall within a configurable similarity threshold. This step produces a set of previously stored interaction summaries that are potentially related to the feedback the user intends to remove.

All retrieved candidates are then provided to the `Retrieve Task Feedback LLM` instance. Rather than relying solely on embedding similarity, this instance performs a semantic selection step by analyzing the internal fields of each candidate entry, with particular focus on the feedback content. Based on this reasoning, the LLM returns the identifiers

of the entries that should be deleted. The configuration of this LLM instance adopts *minimal reasoning effort*, since the operation involves short semantic comparisons rather than complex planning. This design choice reduces latency during feedback-management interactions, ensuring that deletion of task-related preferences does not introduce delays in the interactive control loop.

5.4.6. Update Universal Feedback LLM Instance

The `Update Universal Feedback LLM` instance manages *universal user preferences*, behavioral rules that influence the system across all tasks rather than being tied to a specific execution.

This instance is triggered by the `Plan LLM` through the `change_universal_feedback` low-level skill whenever a user message is interpreted as expressing a global preference or as modifying an existing one. When the `change_universal_feedback` skill is executed, the controller retrieves the current universal preference summary associated with the active user from persistent storage and constructs the user prompt for the `Update Universal Feedback LLM`. The prompt includes both the new user message and the existing preference summary. The LLM instance integrates newly expressed preferences into the summary and removes or revises existing entries whenever the user explicitly or implicitly requests their modification.

For example, consider a user request such as *"Do not notify me anymore when a user is completed"*, combined with an existing universal preference stating *"Include a final log message and rotation movement feedback at the end of each user task"*. The instance updates the summary by removing the logging preference while preserving unrelated behaviors, producing an updated `preferences_summary` that reflects only the remaining valid global preferences.

The configuration of this LLM instance adopts *minimal reasoning effort*, since the operation is limited to semantic interpretation and controlled summarization of user-defined preferences rather than complex inference or multi-step planning. This choice reduces latency during preference-update interactions and prevents global feedback management from introducing delays in the interactive control loop.

5.4.7. Choose Direction LLM Instance

The `Choose Direction LLM` instance implements the exploration policy used by `DACS` to actively navigate the environment when the system must acquire new information

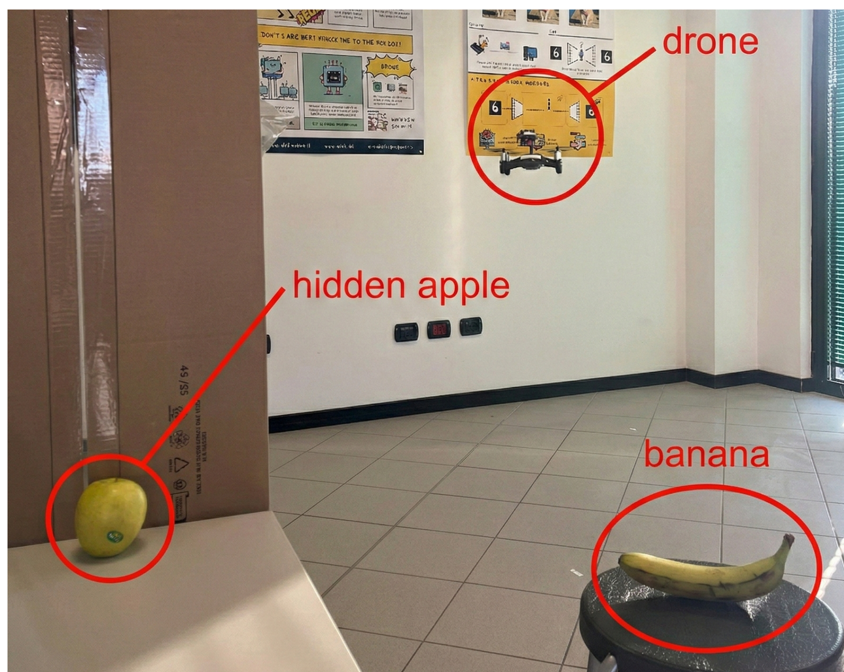


Figure 5.5: External view of the evaluation environment during a *find an apple* task. The yellow apple is occluded behind a vertical structure, while a banana is clearly visible in the scene. In this configuration, the target object is not directly detectable from the current viewpoint, requiring exploration beyond immediate perception. Based on visual context, execution history, semantic associations, and spatial constraints, the `Choose Direction LLM` generates a structured navigation decision (e.g., `yaw = 0`, `distance = 150 cm`, `region_name = "office"`), reasoning that areas containing food-related objects represent semantically promising exploration directions.

to progress toward task completion. Its responsibility is to select a single movement command, defined by a yaw angle and a translation distance, that best advances the current task while respecting safety, feasibility, and exploration constraints. The instance is invoked exclusively through the `explore_direction` low-level skill.

The `Plan LLM` triggers the `Choose Direction LLM` instance when it determines that the user's task cannot be fulfilled within the currently explored region. This situation typically arises when perception does not reveal task-relevant objects in the current scene or when the execution history indicates that repeated local actions are unlikely to produce new information. In these cases, the `Plan LLM` explicitly initiates spatial exploration by inserting the `explore_direction` skill into the MiniSpec plan.

Directional images provided to the `Choose Direction LLM` instance are captured as part of a preceding scanning routine. As specified in the system prompt of the `Plan LLM`,

the `explore_direction` skill may only be invoked after a scan-like procedure has been executed. This procedure rotates the drone at fixed yaw intervals and captures images covering the full 360-degree field around the current position. As a result, when `explore_direction` is executed, it is guaranteed that a complete and up-to-date set of directional images is available for analysis.

When the `explore_direction` skill is executed, the controller constructs the input for the `Choose Direction LLM` instance by aggregating a structured snapshot of the runtime context. This input includes the directional images acquired at fixed 45-unit yaw intervals around the drone, the current task description, the active flyzone definition, the current context graph, and the execution history accumulated during the task. Directional images are explicitly labeled by their corresponding yaw values, providing a consistent geometric reference frame for navigation decisions.

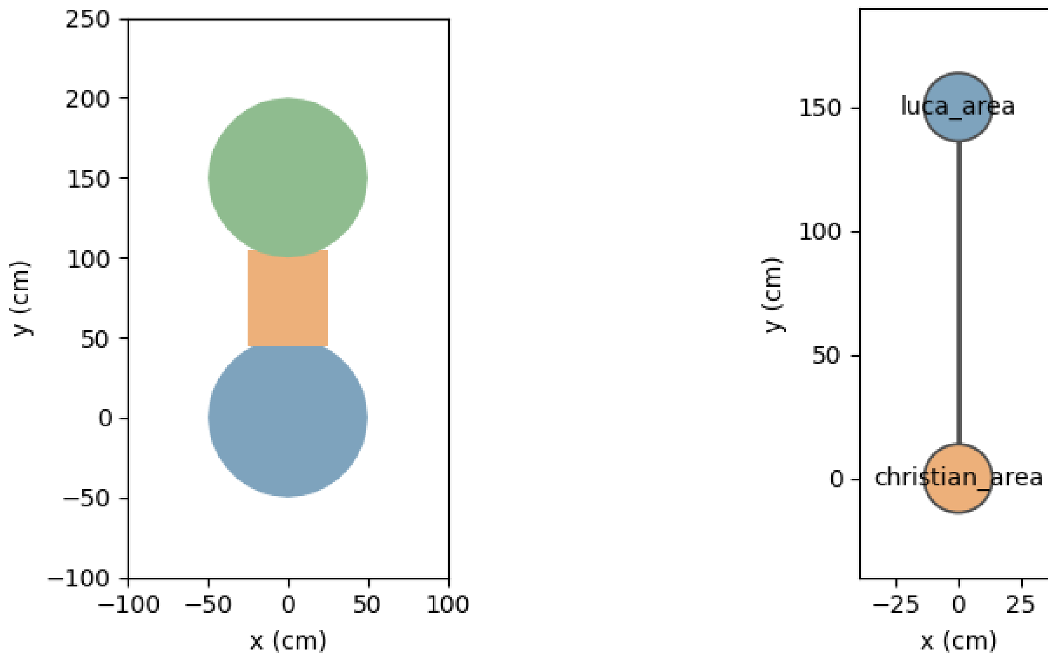
In addition to direction, the instance determines a movement distance. The policy favors distances that produce a meaningful change in context, typically on the order of 100 to 200 cm, while ensuring that the entire movement remains inside the flyzone and is consistent with visible free space.

As part of its output, the `Choose Direction LLM` instance also infers a semantic label for the current region based on the available directional images. This label provides a concise, human-interpretable description of the environment and is used by the controller to update the context graph, allowing regions to be named incrementally as exploration progresses.

The configuration of this LLM instance uses *low reasoning effort*. While adopting minimal reasoning effort would further reduce runtime latency, the task performed by the `Choose Direction LLM` requires a higher level of deliberation than simple summarization or semantic filtering. The instance must jointly reason over multimodal directional observations, the context graph, execution history, and geometric constraints encoded by the flyzone to select an exploration action that is both task-relevant and safe. As this decision directly influences navigation behavior and exploration efficiency, it constitutes a core reasoning component of the system.

5.4.8. Create Graph LLM Instance

The `Create Graph LLM` instance constructs a structured representation of the environment in the form of a *context graph*. The instance is invoked exclusively through the `create_graph` low-level skill and implements the environment modeling capabilities described in Sections 4.3.4 and 5.3.



(a) Geometric flyzone representation generated by the `Create Flyzone LLM`.

(b) Context graph produced by the `Create Graph LLM`.

Figure 5.6: Initializing environment knowledge from a real user-provided description: *"I want to describe the environment where you will execute the plans. The area is composed of two circular regions. The first circle is centered at (0, 0) with radius 50 cm and is named christian_ area. The second circle is centered at (0, 150) with radius 50 cm and is named luca_ area. The two circles are connected by a corridor of width 50 cm."*

The `Plan LLM` triggers this instance whenever a user provides an explicit environment description or uploads visual material depicting the scene layout. In these situations, the `Plan LLM` inserts a `create_graph` call into the MiniSpec plan and forwards the relevant textual or visual information. To support incremental updates, the controller always includes the current context graph in the prompt, allowing the instance to extend, refine, or revise the existing representation rather than rebuilding it from scratch.

This instance operates with *low reasoning effort*. Unlike components involved in the real-time execution loop, the `Create Graph LLM` is typically invoked during environment initialization or explicit model revision. Consequently, strict latency constraints do not apply. At the same time, the task requires structured reasoning, including interpreting spatial descriptions, identifying regions, inferring connectivity, and generating a consistent graph representation.

Figure 5.6 illustrates a representative initialization scenario. The user provides a natural-

language description of the spatial layout, which the `Plan LLM` converts into a MiniSpec plan invoking both `create_graph` and `create_flyzone`. The `Create Graph LLM` translates the description into a context graph encoding regions, coordinates, and connectivity relations, while the `Create Flyzone LLM` (Section 5.4.9) generates the corresponding geometric flyzone representation. Together, these artifacts establish the semantic and geometric foundations required for subsequent planning and navigation.

The output of the `Create Graph LLM` is a single JSON object containing one of the following fields:

- `context_graph`: a structured representation that conforms to the predefined graph schema;
- `user_clarification`: returned only when essential information is missing, such as incomplete region coordinates or ambiguous spatial relations.

Strict completeness constraints are enforced to ensure the reliability of the generated representation. If critical spatial information is absent or cannot be inferred with sufficient precision, the instance does not produce a context graph and instead returns a `user_clarification` request. This prevents incomplete or inconsistent spatial models from propagating into downstream planning and exploration processes.

5.4.9. Create Flyzone LLM Instance

The `Create Flyzone LLM` instance generates a geometric representation of the flyable area starting from a user-provided description of the environment. The instance is invoked exclusively through the `create_flyzone` low-level skill and realizes the flyzone modeling mechanisms introduced in Sections 4.3.7 and 5.3.

This component complements the `Create Graph LLM`, introduced in Section 5.4.8, by translating natural-language spatial descriptions into geometric constraints. While the context graph encodes semantic structure and connectivity between regions, the flyzone defines the admissible operational space of the drone through a set of polygons that bound motion and exploration. The two representations are produced independently and jointly contribute to the internal model of the environment.

The `Plan LLM` invokes the `Create Flyzone LLM` whenever the user provides an explicit description of spatial boundaries or requests an update to the existing environment model. To support incremental refinement, the controller includes the previously stored flyzone in the prompt, allowing the instance to extend or revise the geometric structure without reconstructing it from scratch.

The interaction pattern follows the same design principles adopted for context graph generation. In particular, strict completeness constraints are enforced: if essential geometric parameters are missing or ambiguous, the instance returns a `user_clarification` field instead of producing a partial flyzone. This prevents inconsistent spatial constraints from propagating into navigation and planning.

Figure 5.6 shows a representative initialization scenario in which a natural-language description of the environment leads the `Plan LLM` to invoke both `create_graph` and `create_flyzone`. The resulting flyzone provides the geometric counterpart to the semantic context graph, together forming the environmental knowledge later injected into planning and reasoning.

6 | Prototyping

This chapter describes the physical and computational platform used to prototype and experimentally validate `DACS`. The proposed setup integrates an aerial platform, an external localization module, and an edge computing workstation within a distributed architecture that separates real-time flight execution from perception, reasoning, and planning processes. The following sections present an overview of the hardware configuration and detail the role of each component in supporting the implementation and evaluation of the system.

6.1. Overview

The hardware platform supporting `DACS` is organized to decouple real-time flight execution from computationally intensive perception and reasoning. In this architecture, the aerial platform is responsible only for actuation, image acquisition and localization, while planning, memory management, and execution orchestration are handled offboard on an edge server. The setup consists of three main elements: a `DJI Tello` quadrotor that provides the RGB video stream and executes motion primitives, a `Bitcraze Crazyflie 2.1` equipped with a Lighthouse positioning deck to supply accurate indoor pose estimates, and a Linux-based edge workstation that integrates all software modules and coordinates communication with both devices. All language model inference is performed remotely via API calls, while the edge server remains responsible for user prompt construction, response parsing, and execution of the generated plans.

6.2. DJI Tello Drone

The DJI Tello quadrotor is employed as the primary aerial platform for physical interaction with the environment in `DACS`. Tello is a compact and lightweight consumer-grade drone specifically designed for educational and entry-level research use. Its small form factor, low cost, and robust flight stabilization make it particularly well suited for controlled indoor experimentation, where safety, ease of deployment, and repeatability are critical.

Figure 6.1: Hardware platform of `DACS`. The `DJI Tello` quadrotor provides actuation and visual sensing, while a `Bitcraze Crazyflie 2.1` equipped with a Lighthouse positioning deck is rigidly mounted on top of the drone and supplies accurate indoor localization.

The drone has a total mass of approximately 80 g, which allows safe operation in close proximity to users and objects while reducing the risk associated with accidental collisions.

The drone is equipped with an integrated RGB camera capable of capturing images and streaming video at a resolution of 720p and a frame rate of 30 Hz over a WiFi connection. Under nominal conditions, the video transmission range reaches up to approximately 100 m in open environments, which is more than sufficient for the indoor laboratory scenarios considered in this work. Video transmission and command communication share the same wireless interface and are handled directly by the onboard system through the Tello SDK. The camera stream constitutes the sole perceptual input originating from the drone itself and serves as the basis for all vision-based perception within the system.

From a control perspective, the DJI Tello exposes a deliberately constrained set of high-level motion primitives through its software development kit. These primitives include translational movements along the horizontal plane and vertical axis, with command ranges between 20 and 500 cm, as well as yaw rotations between 1 and 360 degrees. Motion commands are issued in an open loop fashion and executed onboard by the drone's internal

flight controller. Communication between the edge server and the drone is realized through a UDP connection over WiFi, which provides sufficiently low latency for interactive control while remaining simple to integrate and robust to transient packet loss.

Internally, the DJI Tello integrates multiple sensors to support stable flight, including an inertial measurement unit, a barometer, and a downward-facing vision positioning system used for optical flow based stabilization and hover control. These sensors are processed entirely onboard and are not directly accessible to the external control software beyond their effect on flight stability. Power is provided by a single cell lithium polymer battery with a capacity of approximately 1100 mAh, enabling flight times of up to about 13 minutes under nominal conditions. Battery state is monitored onboard and exposed through the SDK, allowing the edge server to enforce conservative safety constraints during execution.

6.3. Crazyflie and Lighthouse Positioning System

Reliable and accurate localization is a key requirement for spatial reasoning, navigation, and context graph maintenance in `DACS`. The onboard sensing capabilities of the DJI Tello are insufficient to provide absolute pose estimates with the precision required for these tasks in indoor and GPS denied environments. To address this limitation, the system integrates an external localization module based on the Bitcraze Crazyflie 2.1 nano drone equipped with a Lighthouse positioning deck. The availability of precise global pose estimates also enables runtime region creation in the context graph; in the experimental setup adopted in this work, a new region is instantiated when the drone moves more than 50 cm from existing region centroids, a value chosen to match the scale of the indoor laboratory environment and the accuracy of the Lighthouse system.

In the proposed setup, the Crazyflie is not operated as an independent flying robot. Instead, it is rigidly mounted on top of the DJI Tello using the protective cage shown in Figure 6.1 and is repurposed exclusively as a localization sensor module. To reduce weight and eliminate unnecessary actuation components, the Crazyflie motors and propellers are removed. This configuration allows the Crazyflie to provide high quality state estimation without affecting the flight dynamics of the underlying drone.

The Crazyflie 2.1 features a dual microcontroller architecture. A STM32F405 microcontroller with a Cortex M4 core running at 168 MHz is responsible for sensor acquisition and state estimation, while a secondary nRF51822 microcontroller handles radio communication and power management. The platform integrates a six axis inertial measurement unit and a high precision barometric pressure sensor, which are used for onboard sensor

fusion.

Global localization is provided by the Lighthouse positioning system. This system relies on four SteamVR base stations placed at fixed locations in the environment, which emit structured infrared light sweeps across the flight volume. The Lighthouse positioning deck mounted on the Crazyflie includes four infrared receivers positioned at its corners, which detect these sweeps and enable the estimation of three dimensional position and yaw orientation within a global coordinate frame. The system operates within a calibrated volume of approximately $5 \times 5 \times 3$ meters, which is sufficient for the indoor laboratory scenarios considered in this work.

6.4. Edge Server

The edge server is responsible for perception, planning, memory management, and execution orchestration, while the aerial platform is dedicated exclusively to sensing and actuation. Although language model inference is performed remotely through API calls to cloud-based services, the edge server manages all intermediate processing stages, including prompt construction, multimodal input encoding, response parsing, and the integration of model outputs into the execution pipeline. In the experimental setup used in this work, the edge server is a laptop workstation running Ubuntu 24.04.3 LTS on a 64 bit architecture. The system is equipped with a 12th generation Intel Core i7-12700H processor, featuring a hybrid architecture with multiple high performance and efficiency cores, and 16 GB of main memory. This configuration provides sufficient computational resources to support real time image processing, concurrent background tasks, and interactive planning without introducing perceptible latency in the control loop.

The workstation includes both integrated Intel Iris Xe graphics and a dedicated NVIDIA GeForce RTX 4050 Laptop GPU. While the proposed system does not rely on GPU acceleration for language model inference, which is performed remotely, the availability of a dedicated GPU is leveraged during development and experimentation for accelerated computer vision pipelines and optional local perception workloads.

The edge server maintains simultaneous communication channels with the aerial platform and the external localization module. Video frames are streamed from the DJI Tello over a WiFi connection, while pose estimates from the Lighthouse positioning system are received through a USB-connected Crazyradio dongle using the `Crazy RealTime Protocol (CRTP)` [8], a lightweight packet-based communication layer designed for low-latency interaction with Crazyflie devices. The server synchronizes these heterogeneous data streams and exposes a consistent and time aligned system state to higher level components.

7 | Evaluation

This chapter presents the evaluation of DACS, the adaptive and context-aware human-drone interaction system proposed in this thesis. The evaluation investigates whether the system effectively supports natural interaction and, crucially, whether it enables users to progressively customize their own interaction interface through use.

The proposed system is evaluated through a comparative user study against the baseline system TypeFly. The comparison is designed to highlight differences in interaction quality, perceived autonomy, and task execution behavior, with particular attention to the adaptive mechanisms introduced in DACS, including environmental knowledge, feedback, shortcuts, and context-aware exploration.

7.1. Goals and Research Questions

The evaluation investigates whether the proposed system enables a form of human-drone interaction that is adaptive, context aware, and user centered. To structure this analysis, the research questions are organized into three complementary dimensions: user adaptation, context awareness, and overall user experience.

- **RQ1 (User Adaptation):** Does DACS enable users to progressively shape and personalize their interaction interface through mechanisms such as feedback rules, flyzones, shortcuts, and high level skills?
- **RQ2 (Context Awareness):** Does DACS effectively exploit contextual information, including the context graph, flyzone, and semantic associations, to influence planning and exploration behavior compared to the baseline TypeFly?
- **RQ3 (Overall User Experience):** Does DACS improve perceived usability, clarity, safety, and overall interaction quality compared to TypeFly?

The remainder of this chapter follows the progression of the evaluation process. It begins with an overview of the study design and procedure (Section 7.2). Participant recruitment and pre-test profiling are then presented in Section 7.3. This is followed by a description

of the experimental setup and interaction protocol, including the physical environment, interaction modalities, and structured scenarios (Section 7.4).

The chapter subsequently introduces the evaluation metrics (Section 7.5), organized to assess the three research dimensions defined above: user adaptation, context awareness, and overall user experience. Each dimension is examined through a combination of user-reported measures, interaction log analysis, and quantitatively extracted metrics.

7.2. Overview

The evaluation adopts a *between-subject* [18] comparative design. Participants are divided into two independent groups, each interacting with only one system. Each group comprises five users, for a total of ten participants: one group interacts exclusively with the baseline system `TypeFly`, while the other interacts exclusively with the proposed system `DACS`.

This choice avoids learning effects and transfer bias that could arise if the same participant interacted with both systems. In particular, we implement `DACS` as an extension built on top of `TypeFly`, sharing the same execution pipeline and interaction backbone. As a consequence, exposing participants to both systems could bias their behavior and expectations in the second interaction, making it difficult to isolate the effects of the adaptive mechanisms from familiarity with the underlying system. By assigning each participant to a single system, we allow users to form a consistent mental model of one interaction paradigm without being influenced by prior exposure to its shared foundations.

From a methodological perspective, a between-subject design is generally considered more conservative, as it limits carry-over effects, order effects, and experimenter demand effects that may emerge when participants are exposed to multiple treatments [18]. In contrast, within-subject designs, while often offering greater statistical power and better control over individual heterogeneity, may introduce contextual comparison and sensitization effects, since participants implicitly contrast the different conditions they experience [18]. In our setting, exposing participants to both systems could lead them to directly compare the two interaction paradigms. This comparison may influence their behavior, making it harder to attribute observed differences solely to the adaptive mechanisms. Although between-subject designs typically require larger samples to achieve comparable statistical power, controlling for cross-condition contamination is particularly important given our limited sample size. To mitigate potential group-level biases, participants were selected to ensure comparable background characteristics across the two groups, including prior drone experience and technical familiarity.

All evaluation sessions follow a *clean slate* principle, meaning that each interaction starts from an empty system state. In particular, for `DACS`, no contextual memory, feedback rules, shortcuts, or high-level skills previously defined by the LLM are present at the beginning of each session. This prevents knowledge accumulated in earlier interactions from influencing subsequent users. As a result, any adaptation or customization observed during interaction emerges exclusively from the participant's own actions within that session, rather than from pre-existing state.

7.3. Participants

A total of ten participants take part in the study. Following the between-subject design described in Section 7.2, participants are divided into two independent groups of five users each. One group interacts exclusively with `TypeFly`, while the other interacts exclusively with `DACS`.

Participants are recruited on a voluntary basis and exhibit heterogeneous technical backgrounds, as well as varying familiarity with drones and intelligent interfaces. Most participants have a technical or engineering profile. To ensure representation of non-expert users, each group includes at least one participant without a technical background.

7.3.1. Pre-Test Profiling

Before starting the interaction session, all participants complete a pre-test questionnaire designed to collect demographic information and to assess prior experience, familiarity with relevant technologies, and initial expectations regarding the system.

The collected data provide contextual information that supports the interpretation of user behavior and performance during the evaluation. The complete pre-test questionnaire is reported in Appendix A for reproducibility.

Table 7.1 summarizes the pre-test profiles of the participants assigned to the two systems. Across the ten participants, most users have a technical or engineering background, while each group includes at least one non-technical participant. This composition ensures that the evaluation reflects both technically confident users and individuals with less specialized expertise, allowing observation of diverse interaction strategies.

Direct experience with drone control is generally limited in both conditions. Most participants report little or no prior piloting experience, meaning that interactions primarily reflect first-time or occasional users rather than trained operators. This choice aligns with the intended target audience of natural-language drone interfaces, which are designed for

User	Age	Tech	Drone exp.	AI fam.	NL fam.	Main concern
DACS participants						
U1	28	Yes	Few	5	4	None
U2	27	Yes	Few	5	4	Request understanding
U3	25	Yes	None	4	4	Autonomy
U4	24	No	None	5	3	Safety
U5	24	No	Few	4	4	Exploration
TypeFly participants						
U6	25	Yes	Few	5	3	Exploration
U7	58	Yes	None	4	3	Safety
U8	24	Yes	None	5	5	Autonomy
U9	24	No	None	5	4	Exploration
U10	24	No	Few	4	4	Request understanding

Table 7.1: Pre-test profiles of participants in both conditions. *Tech* indicates a technical or engineering background. *AI* denotes familiarity with Artificial Intelligence systems, and *NL* denotes familiarity with Natural Language interfaces, both measured on a 1-5 Likert scale (1 = none, 5 = very high).

accessibility rather than expert control.

In contrast, familiarity with conversational and AI-based interfaces is consistently moderate to high across all participants. Users are therefore already accustomed to interacting with intelligent systems through natural language. As a result, difficulties observed during the evaluation are more likely attributable to system behavior and design choices rather than to unfamiliarity with the interaction modality itself.

Reported expectations and concerns are also similar between groups. Participants typically expect reliable low-level navigation commands combined with a certain degree of autonomy, while expressing concerns related to safety and correct command interpretation.

Overall, the similarity between the two populations supports a fair between-subject comparison. Differences observed in the evaluation can therefore be interpreted primarily as consequences of the systems' capabilities rather than demographic or experiential imbalances between groups.

7.4. Evaluation Protocol

This section describes the evaluation protocol adopted to assess both `TypeFly` and `DACS`. It details the physical environment, interaction modalities, and structured interaction scenarios used during the sessions. The protocol is designed to ensure controlled and comparable conditions across participants while progressively increasing task complexity

Figure 7.1: Indoor office environment used during the evaluation.

to observe usability, adaptation, and behavioral evolution over time.

7.4.1. Physical Environment

The evaluation is conducted in a dedicated indoor area within an office environment, specifically allocated for drone experimentation and user interaction. The environment provides a controlled yet realistic setting for human-drone interaction tasks, while ensuring safety, repeatability, and consistent operating conditions across sessions.

The experimental space consists of a bounded planar area, defined as a square of side 2.5 m, with a vertical extent of 3 m. The environment is internally represented through a three-dimensional coordinate system used for navigation and localization. Figure 7.1 shows the indoor environment used during the evaluation. The scene includes common office furniture and everyday objects (e.g., bottles, fruit, and small items placed on tables or surfaces), which serve as targets for perception, navigation, and search tasks.

While the overall size and structure of the environment remain fixed throughout the evaluation, the number, type, and spatial arrangement of objects vary across scenarios. This variability is intentional and enables testing under progressively more complex conditions, including simple object discovery, active exploration, and context-aware navigation. The specific object configurations adopted in each scenario are described in detail later in the chapter.

7.4.2. Interaction Modalities

During the evaluation, both `TypeFly` and `DACS` are accessed through a web based interface that provides a chat component for natural-language interaction. Users issue instructions by typing text commands in the web interface, and system responses are returned in textual form. This design choice ensures uniformity in the interaction modality across systems and isolates the effects of the adaptive mechanisms from differences in input or output channels.

In the case of `DACS`, the web interface is extended with additional visualization components. In particular, users can visualize the currently active flyzone, making spatial constraints explicit and understandable. This allows users to remain aware of flyzone boundaries and to request modifications or the creation of new flyzones through natural-language commands. The same interface also provides a visualization of the current context graph, including the estimated drone pose and the set of regions explored during the interaction, together with the objects associated with each region. While interaction and control remain entirely chat based, these visual elements support user awareness of the system internal state.

7.4.3. Interaction Scenarios

Each evaluation session is organized as a sequence of *open-ended interaction scenarios*. Rather than prescribing detailed step-by-step instructions, participants are given high-level goals (e.g., search for objects) and are then free to interact with the system autonomously within that context.

This design choice is intentional and aligns with interaction design perspectives that view openness and interpretative flexibility as valuable resources for understanding how users appropriate and make sense of interactive systems [28]. The objective of the evaluation is to observe how users naturally formulate commands, explore available functionalities, and adapt their interaction strategies without strict procedural constraints. By avoiding scripted tasks, the protocol aims to capture spontaneous behavior, realistic usage patterns, and individual differences in learning and problem solving. This approach allows the evaluation to assess not only task performance, but also intuitiveness, learnability, and feature discoverability.

At the beginning of each scenario, participants receive a brief description of the goal and are given several minutes to freely use the system. During this time, no specific interaction steps are enforced. This free interaction phase enables observation of how quickly users

understand the interface, how they construct commands, and whether they independently discover advanced features such as memory mechanisms, feedback rules or flyzone.

The open-ended structure also provides insight into whether system functionalities are used as intended or repurposed in unexpected ways, which is particularly relevant for adaptive and customizable systems such as `DACS`. Such observations support a more comprehensive evaluation of both usability and system design assumptions.

Scenario 0: Self-Guided Tutorial and Familiarization

Duration: approximately 10 minutes.

This preliminary scenario serves as a self-guided tutorial phase in which participants independently familiarize themselves with either system. Rather than providing a structured demonstration or explicit training, users are encouraged to discover the system's capabilities through direct experimentation using the chat-based interface.

During this phase, no real drone is deployed. Instead, the system operates with a virtual drone represented within the laptop interface. Removing the physical platform intentionally eliminates the operational complexity, safety considerations, and attentional demands associated with flight control, allowing participants to focus exclusively on understanding the interaction paradigm and available functionalities.

Participants receive only the following high-level instructions:

> *"Use the system freely to explore what the drone can do. Try different commands, ask questions, and experiment with its capabilities. If you are unsure, you may ask the system for help or for a description of its available functions."*

No further guidance is provided. The purpose of this scenario is to allow users to freely explore the system and construct their own understanding of its capabilities before performing operational tasks.

Scenario 1: Object Search and Repeated Task Execution

Duration: approximately 10 minutes.

Environment: open indoor area with multiple everyday objects placed in plain sight on tables (e.g., fruit, bottles, bags), as illustrated in Figure 7.1. All targets are directly visible and reachable, and no occlusions or complex navigation constraints are introduced.

This scenario represents the first interaction with the physical drone and transitions from the virtual familiarization phase to embodied operation. The objective is to evaluate how

participants issue concrete task requests in a simple and controlled setting, focusing on basic navigation, perception, and task formulation through natural language.

Participants are asked to use the drone to locate objects distributed in the environment. Tasks begin with simple requests (e.g., "find an apple") and progressively evolve toward repeated and slightly more complex queries. Users are encouraged to search for multiple objects and to repeat the same request at different times during the session. Participants receive only the following high-level instructions:

> *"Use the drone to help you find any objects in this area. Start with simple requests, then try repeating or modifying them to better match what you need. You may search for different objects or ask for the same one multiple times."*

The progressive and repetitive nature of the tasks is designed to encourage natural refinement of interaction strategies. As users gain confidence, they may reformulate commands, provide corrections, or express higher-level intentions rather than issuing only low-level navigation directives.

This scenario primarily exercises:

- basic drone navigation,
- early corrective feedback (e.g., adjustments to how tasks are executed),
- repeated task execution to assess the use of persistent memory and the context graph,
- the creation and reuse of higher-level skills,
- the definition and use of user-specified spatial constraints (flyzones) to restrict or guide motion.

Scenario 2: Object Search with Exploration and Partial Observability

Duration: approximately 10 minutes.

Environment: the same room layout as Scenario 1, but with partial occlusions. Some target objects are intentionally hidden behind furniture and are not directly visible from the initial drone position, requiring active exploration by the drone. For example, an apple may be placed near a visible banana to create a semantically coherent area that can support contextual reasoning.

Participants receive the same open-ended instructions as before and are encouraged to explore freely when a target is not immediately visible.

The additional aspects evaluated in this scenario are:

- reasoning over partial observations and contextual cues (e.g., searching near semantically related objects),
- incremental construction and use of the context graph across multiple regions,
- the use of flyzone to constrain or structure exploration strategies.

This setup therefore stresses the system's ability to move beyond reactive perception and to support informed, experience-aware exploration. The context-aware mechanisms of `DACS` are expected to provide advantages in this setting, whereas the baseline `TypeFly` relies primarily on stateless replanning and immediate observations.

Scenario 3: Unconstrained Interaction and Complex Task Composition

Duration: approximately 10 minutes.

Environment: same indoor setting with multiple objects distributed across the room. Additional variability may be introduced during the session, such as adding new items or the presence of a moving person. Participants are explicitly allowed to modify the environment if desired, including moving existing objects, introducing new objects, or changing their own position within the space.

This final scenario removes all task constraints and follows the structured activities of the previous phases. Having already gained familiarity with the interface and system capabilities, participants are free to interact with the drone in any way they choose and to define their own goals. The objective is to observe how the system is used in a fully open-ended setting, without predefined tasks or suggested behaviors.

Participants receive only the following instructions:

> *"Anything you want to try is fine. You can freely experiment with the drone, and you may also modify the environment by moving objects or introducing new ones."*

Compared to the previous scenarios, this phase introduces complete freedom in both task definition and environmental configuration. This setting is intended to encourage more proactive and creative interaction, potentially leading users to formulate longer or more

complex tasks, combine multiple objectives, or request behaviors that were not explicitly exercised before.

The additional aspects evaluated in this scenario are:

- autonomous use of shortcuts,
- progressive personalization through feedback, memory, and spatial constraints,
- robustness of the system under unpredictable or dynamically changing conditions.

This unconstrained interaction phase provides insight into how the systems are adopted in practice once users are no longer guided by experimental prompts. In particular, it allows assessment of whether the adaptive mechanisms of `DACS` naturally support more efficient and personalized workflows, such as packaging frequently repeated behaviors into reusable shortcuts, compared to the stateless baseline `TypeFly`.

7.5. Results

This section presents the results of the evaluation based on the defined metrics, assessing both system behavior and user experience. For each evaluated capability, we first describe the aspect under investigation and then report the corresponding outcomes.

Metrics are grouped into three complementary categories:

- **User-reported metrics**: measures collected through the post-test questionnaire, capturing subjective user perception. For completeness, the full questionnaire can be accessed at: `https://tinyurl.com/DACSresults`;
- **Interpretation through interaction logs**: qualitative analysis of system behavior based on execution traces and interaction logs;
- **Log-retrieved metrics**: quantitative measures extracted from system logs, such as timing, counts, and execution statistics.

This structure ensures that each capability is evaluated from multiple perspectives, combining subjective user feedback with objective evidence derived from observed system behavior.

7.5.1. System Self-Explainability

Evaluation goal. *System self-explainability* refers to how easily users can understand what the system is capable of while interacting with it. In human-drone interaction,

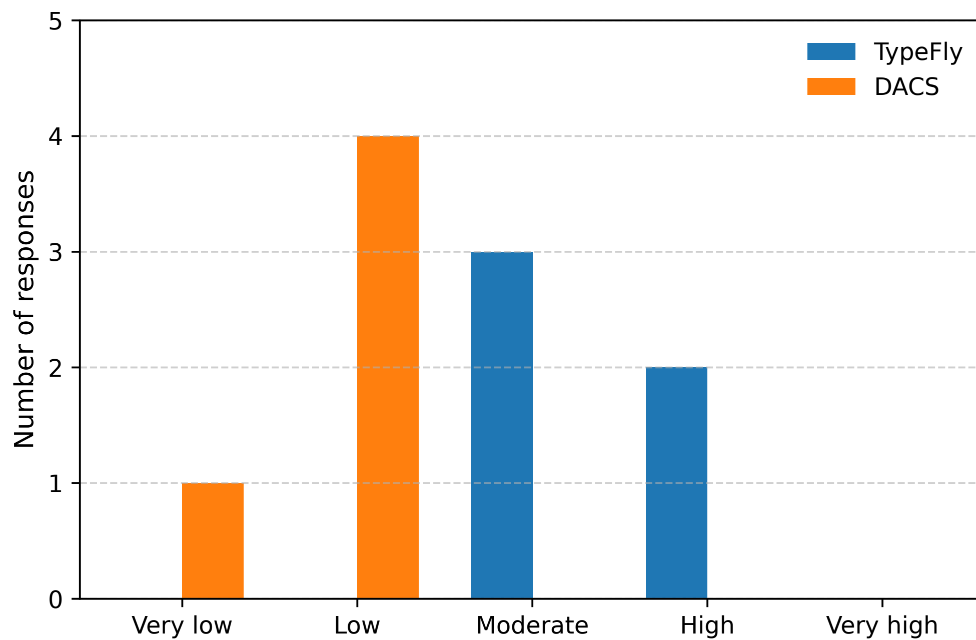


Figure 7.2: User-reported difficulty in understanding system capabilities. Responses are reported for both `TypeFly` and `DACS`.

this property is fundamental: if system capabilities are not clear, users may struggle to formulate appropriate requests, rely on trial-and-error interaction, or develop inaccurate mental models of the system's autonomy.

This aspect is particularly relevant during Scenario 0 (Section 7.4.3), when users actively explore the system and attempt to infer its capabilities through dialogue. A self-explanatory system should therefore not only expose its available functions, but also guide users toward correct and effective ways of expressing their intent.

User-reported metrics. System self-explainability is evaluated through a user-reported measure collected in the post-test questionnaire. Participants were asked:

> *"How difficult was it to understand what the system was capable of while using it?"*

Figure 7.2 reports the response distributions for both `TypeFly` and `DACS`. The baseline system exhibits predominantly mid-to-high difficulty ratings, whereas `DACS` receives consistently low difficulty scores. This indicates that participants found it significantly easier to understand the system's capabilities when interacting with the adaptive version.

Interpretation through interaction logs. The difference observed in the user-reported data is reflected clearly in the interaction logs collected during Scenario 0. When explicitly asked about its capabilities, `TypeFly` responds with an enumeration of low-level primitives:

```
I can move forward, backward, left, right, up, down, turn clockwise
or counterclockwise, check the visibility of an object, probe for
reasoning, log text, take a picture, go to an object and scan for
an object.
```

Although technically accurate, this response exposes internal actions rather than user-oriented affordances. Capabilities are presented as isolated commands, without explaining how they relate to higher-level goals such as navigation, exploration, or spatial constraint management. The system does not suggest how these primitives can be combined, nor does it guide the user toward valid task formulations. As a consequence, users must infer the system's functional scope through experimentation, which is consistent with the higher perceived difficulty reported in Figure 7.2.

In contrast, `DACS` provides structured and goal-oriented explanations of its capabilities. As described in Section 7.4.3, during the self-guided familiarization phase participants are encouraged to freely ask the system about its functionalities in order to explore its features. When prompted with such questions, the system organizes its functionalities around meaningful interaction concepts:

```
Here is how I can help:
1) Navigate and explore within a defined flyzone.
2) Use an environmental memory to remember regions and objects.
3) Search for objects and go to them.
4) Store preferences and task-specific rules.
5) Create reusable skills and shortcuts.
6) Enforce safety constraints and ask for missing information.
```

This structured response emerges directly from the architectural design of `DACS`. As described in Section 5.4.2, the Plan LLM receives an explicit description of the drone's available skills, which provides a clear overview of what the system can do. Importantly, these skills are not limited to low-level motion primitives, but also include reasoning-oriented skills that internally invoke other LLM instances (Section 5.4). As a consequence, the Plan LLM reasons over a unified skill space that encompasses both physical execution and higher-level cognitive operations such as memory updates, flyzone management, or graph construction. This unified representation enables the system to present its capabil-

ities in structured, goal-oriented terms rather than as isolated commands.

This high-level overview acts as an entry point rather than an exhaustive description. Importantly, DACS allows users to progressively explore each capability in more detail through follow-up questions. Users are not required to understand all abstractions upfront: instead, the system reveals additional constraints, parameters, and valid formulations only when needed, supporting incremental learning during interaction.

This behavior becomes particularly evident when users issue underspecified or ambiguous commands. Rather than rejecting the request or making arbitrary assumptions, DACS actively drives the user toward a well-formed instruction. For instance, during flyzone creation, user U2 issues the request:

```
Change the flyzone to a square with a side length of 100
```

Instead of attempting an implicit interpretation, the system explicitly decomposes the task and requests the missing information:

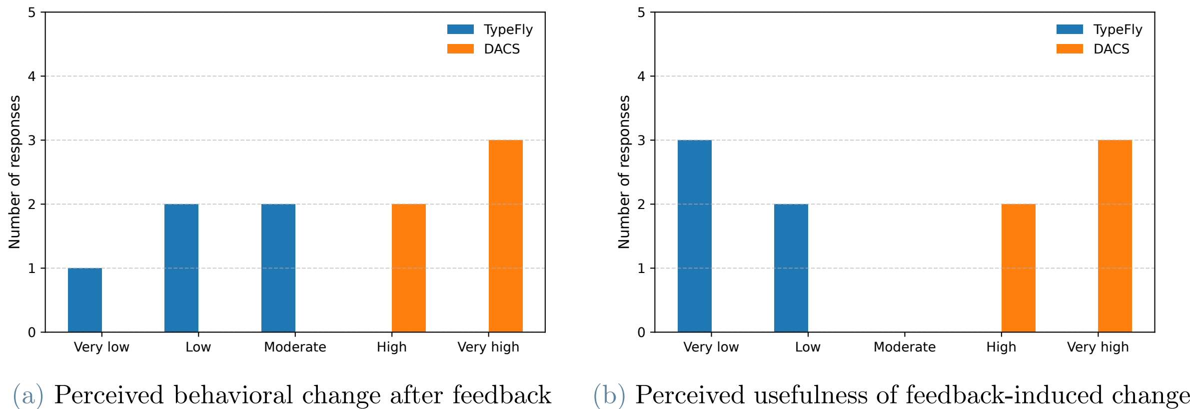
```
To change the flyzone to a square, please confirm:
1) Side length and units (does "100" mean 100 cm?).
2) Exact placement (center coordinates, or a corner point).
3) Orientation (aligned with axes or rotated?).
```

When the user provides an incomplete response, the system does not proceed blindly. Instead, it refines the clarification request and offers a default configuration to reduce user effort:

```
If you want standard defaults, reply "use defaults" and I will set:
side=100 cm, center=(0,0), orientation=0, replace current=yes.
```

After the user accepts the defaults, the system transparently confirms the interpreted action and applies the flyzone update. Throughout this interaction, the system progressively educates the user about which parameters define a flyzone, which assumptions are required, and how ambiguity is resolved. In doing so, DACS does not merely execute commands, but actively scaffolds the interaction, guiding the user toward valid and meaningful requests.

This ability to explain capabilities at a high level, deepen explanations on demand, and drive the user through structured clarification directly supports the lower perceived difficulty observed in Figure 7.2, and highlights a fundamental difference in how the two systems expose and communicate their functional scope.



(a) Perceived behavioral change after feedback (b) Perceived usefulness of feedback-induced change

Figure 7.3: User-reported feedback effectiveness for `TypeFly` and `DACS`. Each subplot corresponds to one questionnaire item, with colors distinguishing the two systems.

7.5.2. Feedback Mechanisms

Evaluation goal. Feedback mechanisms constitute one of the core adaptive capabilities introduced in `DACS`. The objective of this evaluation is to assess whether users can explicitly refine system behavior through natural-language corrections and whether these refinements are correctly interpreted, stored, and reused across subsequent tasks.

Two complementary aspects are considered:

- **feedback creation**, namely the ability of users to express corrections or preferences in natural language and have them recognized as feedback rather than as new tasks;
- **feedback usage**, namely the correct integration of stored feedback into future task execution.

Since persistent feedback mechanisms are not implemented in the baseline `TypeFly`, the comparison between the two systems highlights the effect of stateful versus stateless interpretation of user corrections.

User-reported metrics. User perception of feedback effectiveness is assessed through two post-test questionnaire items:

1. *"When you corrected or refined your request, how much did the system's behavior change?"*
2. *"Think of a moment when the system changed its behavior after something you said. How useful was that change?"*

Figure 7.3 reports the response distributions for both systems, organized by questionnaire item. For the baseline `TypeFly`, subplot (a) indicates limited and inconsistent adaptation. Two participants (U8 and U9) report a moderate behavioral change (rating 3) immediately after providing a correction. However, subplot (b) shows that the perceived usefulness of that change is rated low (values 1-2). This pattern suggests that, while the system may appear to react to a correction, the effect is not perceived as stable or beneficial over time.

In contrast, responses for `DACS` show a markedly different pattern. All participants report strong behavioral changes (ratings 4-5), and the perceived usefulness of these changes is consistently high. This indicates that users not only observe the effect of their feedback, but also experience tangible benefits from its persistence across tasks.

Interpretation through interaction logs. To explain the user-reported perceptions, evidence of feedback handling is extracted from execution logs by analyzing how the system interprets, stores, and applies user corrections during interaction.

Across all participants interacting with `DACS`, each user issued at least two explicit feedback instructions during the session. In all observed cases, the system correctly recognized that the user intended to provide feedback rather than request a new task. In most cases, the feedback was task-related (e.g., modifying how an object search is performed) and was correctly classified as such.

In one representative case, participant U4 issued the instruction:

> `Great work, from now on whenever you complete a task go up and down vertically.`

This instruction was correctly interpreted as *universal feedback*, affecting all subsequent tasks rather than a specific task category.

Log inspection further shows that, whenever previously defined feedback was relevant, it was:

- inserted into the planning prompt of subsequent tasks,
- applied during execution without requiring repetition,
- reflected consistently in the generated behavior.

This persistent reuse of feedback explains the high usefulness ratings reported by users of `DACS`.

In contrast, the baseline `TypeFly` does not store feedback persistently. In several cases, the system appears to adapt its behavior immediately after a correction by implicitly

interpreting it as part of the current task. However, since the correction is not stored, the behavior reverts in subsequent interactions. This explains why some users report a moderate immediate behavioral change but assign low usefulness scores: the adaptation is transient and does not persist across tasks.

Taken together, the log-based analysis provides a concrete explanation for the observed questionnaire responses, confirming that persistent feedback handling in `DACS` leads to stable and perceivable adaptation, whereas the stateless baseline exhibits only short-lived, task-local adjustments.

7.5.3. Context Graph Memory

Evaluation goal. This section evaluates the system's ability to accumulate, retain, and reuse knowledge about the environment through a persistent memory structure. In `DACS`, this capability is implemented through the *context graph*, which stores explored regions, detected objects, and their spatial relationships. The objective of this evaluation is to determine whether previously acquired information is reused in subsequent interactions and whether this reuse is clearly perceivable by users.

The baseline system `TypeFly` does not maintain any persistent spatial or semantic representation and processes each request independently. It therefore serves as a control condition representing a stateless interaction model.

User-reported metrics. Perceived memory and contextual understanding are assessed through post-test questionnaire items asking participants whether the drone appeared to remember previously visited regions, objects, or contextual information.

Figure 7.4 highlights a clear contrast between the two systems. For the baseline `TypeFly`, responses concentrate at the lower end of the scale: four participants report that the drone does not appear to remember previous regions, objects, or contextual information, while only one participant assigns an intermediate score. This distribution reflects a stateless interaction model in which each task is perceived as starting from scratch, and any apparent continuity arises solely from short-term perceptual availability.

In contrast, responses for `DACS` are consistently high. All participants assign ratings of 4 or 5, indicating a strong perception that the system remembers past observations and reuses them during interaction. This suggests that the presence of a persistent context graph produces behavioral effects that are not only functional but also clearly observable from the user's perspective.

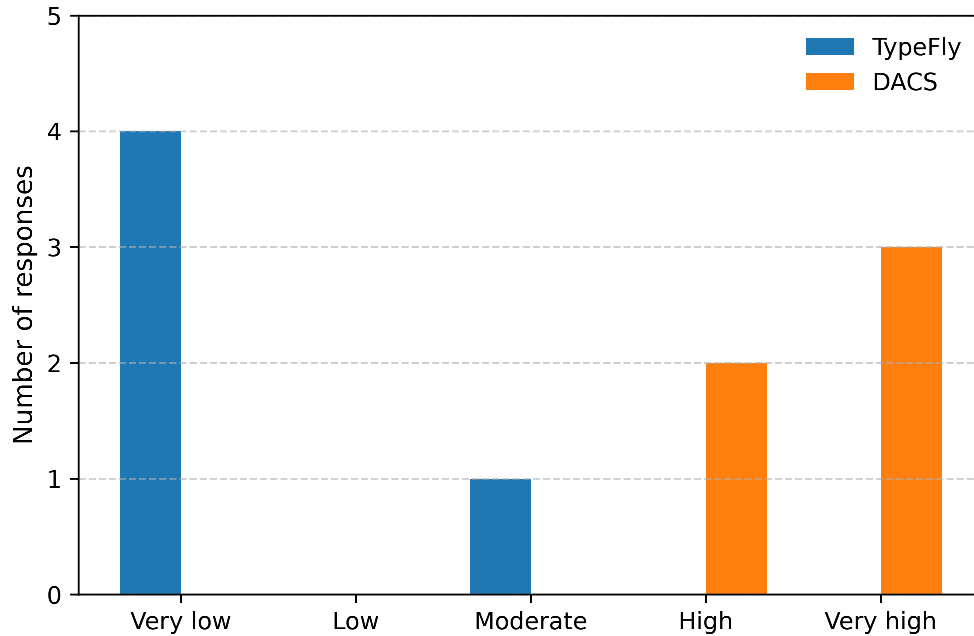


Figure 7.4: User-reported perception of memory and contextual understanding for the baseline `TypeFly` and `DACS`. Participants rated whether the drone appeared to remember previously visited regions, objects, or contextual information.

Interpretation through interaction logs. To justify the user-perceived behavior, evidence of context graph usage is extracted from interaction logs by analyzing exploration decisions, replanning behavior, and explicit user queries related to previously observed information.

At this stage, the evaluation does *not* assess the quality or optimality of the chosen exploration directions. Directional decision quality is analyzed separately in Section 7.5.5. Here, the objective is to verify whether information stored in the context graph is explicitly taken into account during planning, reasoning, and dialogue.

When the system initiates exploration after unsuccessful local perception, the `Choose Direction LLM` instance generates an explicit exploration decision expressed in terms of yaw and distance, accompanied by a natural-language explanation. This explanation is intentionally produced by design, in order to make the reasoning process transparent and inspectable for evaluation purposes. The generated explanations consistently reference contextual information stored in the context graph, such as previously explored regions, object-region associations, or region connectivity, rather than relying solely on immediate perceptual cues.

Context graph usage is not limited to direction selection. Stored spatial and semantic

information is also reused by other LLM instances during different phases of execution. In particular, the `Short-Term Memory LLM` instance incorporates context graph information when summarizing execution outcomes and guiding subsequent replanning steps. For example, during a banana search task, the system generates the following iteration summary:

> `After repeated scans failed to detect a banana, the system concluded that the target was not visible in the current region and triggered exploration, explicitly noting that bananas were previously associated with region_0 in the context graph and suggesting movement toward that region as a next step.`

This summary explicitly references object-region associations stored in the context graph and uses them to inform subsequent planning decisions, demonstrating that contextual knowledge is reused beyond low-level action selection.

Context graph usage is also evident when users explicitly query previously observed information. In one case, participant U3 asks whether the system has already seen bananas and where they were located. The system responds:

> `From prior exploration, bananas were found in region_0 at [79.5, -79.9] and region_2 at [-64.2, -141.4]. The nearest from here is region_2. Would you like me to go there?`

This response directly retrieves stored object-region associations from the context graph, demonstrating that past observations are persistently stored and accessible for reasoning and dialogue.

7.5.4. Flyzones and Safety

Evaluation goal. Flyzones define explicit spatial constraints that restrict the drone to operate only within user-specified regions. The objective of this evaluation is to assess whether users perceive drone motion as safe, predictable, and controllable, and to understand how explicit spatial constraints influence this perception during interaction.

In `DACS`, flyzones are explicitly represented and provided as contextual constraints to the LLM instances responsible for decision-making, including the `Plan` and `Choose Direction LLM` instances. As a result, generated navigation and exploration actions are conditioned on the currently active flyzone. In contrast, the baseline system `TypeFly` does not incorporate any explicit spatial constraints, and the generated plans are not conditioned on a flyzone.

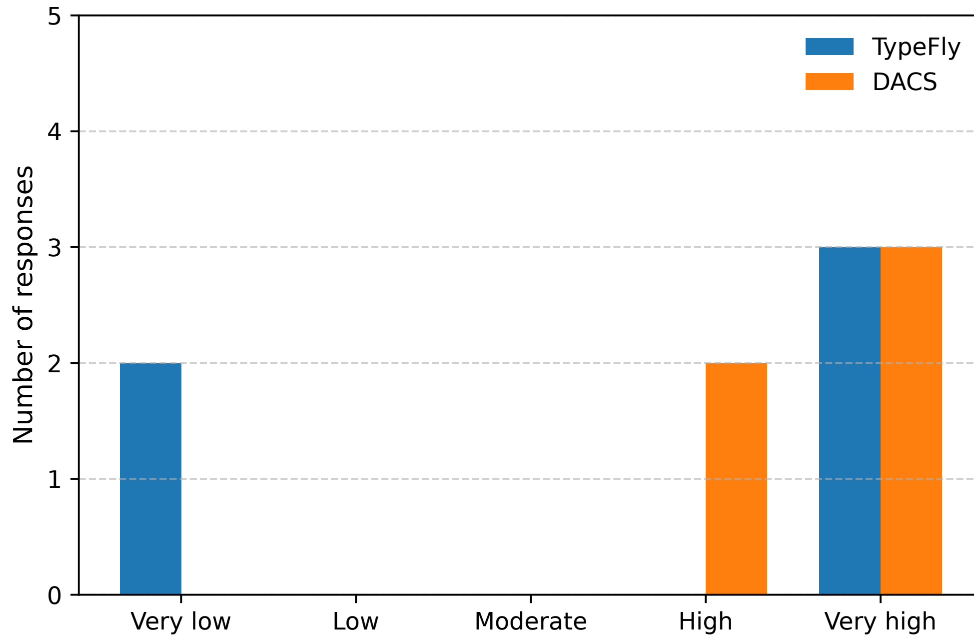


Figure 7.5: User-perceived spatial safety and predictability across systems. Participants rated whether the drone tended to stay within the areas they expected it to be.

User-reported metrics. Perceived spatial predictability and safety are evaluated through a post-test questionnaire item asking participants:

> *"Did the drone tend to stay in the areas where you expected it to be?"*

Figure 7.5 shows a clear contrast between the two systems. For the baseline `TypeFly`, responses are strongly polarized. Some participants assign very low scores, indicating that the drone frequently moves outside expected areas, while others assign high scores. This polarization arises from the nature of the supported tasks: during perception-oriented tasks such as object search, the drone often remains stationary and performs only rotational scanning, which users perceive as safe and predictable. In contrast, for explicit movement commands (e.g., *"go forward by . . . "*), the drone may translate without spatial awareness, potentially exiting the evaluation area and colliding with walls, objects, or people. As a result, user perception varies significantly depending on how the system is used.

In contrast, responses for `DACS` are consistently high. All participants report that the drone reliably stays within expected areas. This indicates that explicit spatial constraints lead to stable and predictable behavior across different task types.

Interpretation through interaction logs. The observed questionnaire responses are explained by inspection of interaction logs, which reveal fundamentally different planning behaviors in the two systems.

In `DACS`, flyzones are enforced directly by the `Plan LLM` instance. When a user requests a target outside the permitted region, the planner clips the trajectory to the nearest valid boundary and informs the user. For example, when the user issues the command:

```
go to -175, 175
```

the system generates a constrained plan and reports:

```
Stopped at [-125, 125] because the requested [-175, 175] is outside
the flyzone.
```

This behavior prevents unsafe motion while making the spatial constraint explicit, directly contributing to the high predictability ratings observed in Figure 7.5.

Flyzones can also be explicitly created and reshaped by users through natural-language commands. For instance, participant U2 requests:

```
Change the flyzone shape to a circle with radius 20 cm
```

The system correctly interprets this instruction as a flyzone redefinition request and produces a concrete geometric representation of the requested region. Evidence of this behavior can be observed directly in the generated plan, which contains the discretized polygon defining the new flyzone:

```
points_list: [[[20, 0], [20, 2], [19, 5], ..., [-20, 3], [-20, 2],
[20, 0]]]
```

This output shows that the circular flyzone requested by the user is internally represented as a closed polygonal approximation, which is then activated as the current spatial constraint. The newly defined flyzone is subsequently enforced during navigation and exploration, demonstrating that flyzones can be dynamically defined through natural language and persistently applied across tasks.

Flyzone awareness is also reflected in exploration behavior. When selecting exploration actions, the `Choose Direction LLM` instance reasons not only about semantic relevance and exploration goals, but also about safety constraints. In one exploration step, the system produces the following justification:

```
The path appears clear and remains well within the flyzone, so 180
cm advances exploration meaningfully.
```

Here, the chosen distance is explicitly justified in terms of remaining inside the flyzone, demonstrating that spatial constraints influence both goal-directed navigation and exploratory motion (see Section 7.5.5) for a detailed analysis of directional reasoning).

In contrast, the baseline `TypeFly` executes motion commands without spatial validation. While this leads to safe behavior in tasks that do not require translation, it can result in large, unconstrained displacements during movement commands. This explains the polarized questionnaire ratings observed for `TypeFly`: depending on task formulation, the drone may appear either safe and stationary or unpredictable and unsafe.

7.5.5. Directional Visual Reasoning

Evaluation goal. This section evaluates how users perceive the quality and purposefulness of the drone's exploration behavior when the target is not immediately visible. The objective is to assess whether navigation decisions appear reasonable and goal-directed from a human perspective, rather than random, passive, or inefficient during search tasks.

In `DACS`, directional decisions are generated by explicitly reasoning over visual cues, semantic associations, and stored contextual knowledge. In contrast, the baseline `TypeFly` does not implement explicit directional reasoning for exploration and relies primarily on local or reactive behaviors.

User-reported metrics. Perceived exploration quality is evaluated through a post-test questionnaire item asking participants:

> *"When the target was not immediately visible, how confident were you that the drone was searching in the right way?"*

Figure 7.6 shows a clear contrast between the two systems. For the baseline `TypeFly`, responses are concentrated at the lower end of the scale, indicating limited user confidence that the drone was searching in an appropriate or meaningful direction. This reflects the fact that, when the target is not visible, the baseline system does not autonomously relocate and instead relies on repeated local scans, often stalling progress unless the user intervenes manually.

In contrast, confidence ratings for `DACS` are consistently high. All participants report values in the upper range of the scale, indicating that the drone's exploration behavior appears intentional and aligned with the task objective. This suggests that explicit directional reasoning produces exploration behavior that is not only functionally effective but also clearly interpretable by users.

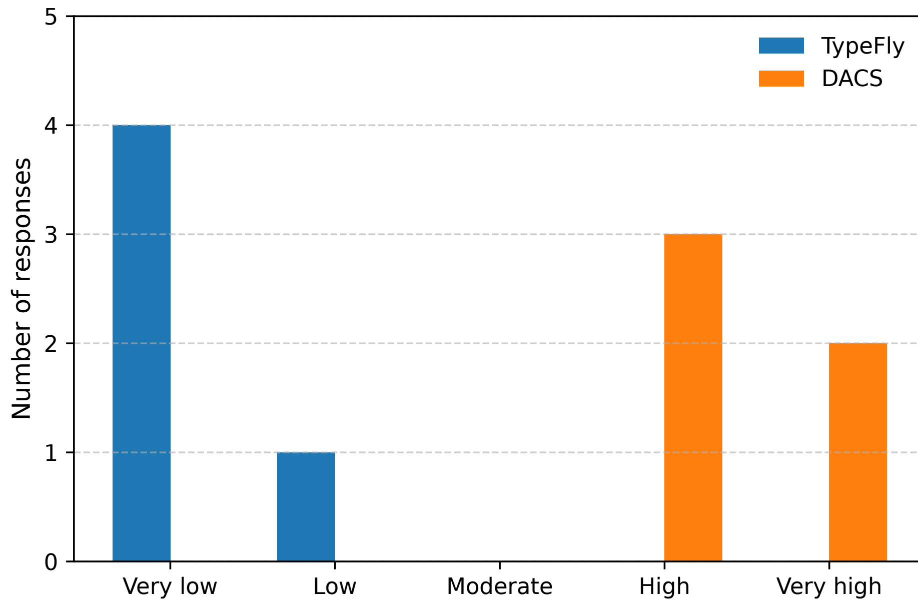


Figure 7.6: User-reported confidence in directional search behavior when the target is not immediately visible. The merged diagram compares the baseline `TypeFly` with `DACS`.

Interpretation through interaction logs. The observed difference in user confidence is explained by inspection of interaction logs, which reveal fundamentally different exploration strategies in the two systems.

In the baseline `TypeFly`, hidden-object search consistently degenerates into repeated local scanning without meaningful translation. When the target object is not immediately visible, the system repeatedly checks visibility from the same viewpoint instead of selecting a new direction to explore. As a result, users are forced to manually guide the drone through the environment using low-level motion commands in order to continue the search. This behavior is illustrated by participant U9, who explicitly takes control of the exploration process through a long sequence of commands such as:

```
fly up by 70 ... go straight for 50 cm ... turn by 180 ... tell me
if you see the backpack
```

The user effectively replaces the system's exploration strategy by manually moving the drone step by step. The need for such intervention explains the low ratings observed in the questionnaire for `TypeFly`: exploration does not appear autonomous or goal-directed, but instead stalls unless the user actively intervenes.

In contrast, `DACS` performs autonomous and context-aware exploration when the target is not immediately visible. Directional decisions are generated by the `Choose Direction`

`LLM`, which reasons over visual affordances, semantic associations, and stored contextual information to select both a direction and a distance for exploration. For example, during an automatic hidden-object search, the system generates the following exploration decision:

> `The 45° view shows open floor leading toward desks and visible food items (bananas), a task-relevant area likely to also contain fruit like an apple. The path appears clear and remains well within the flyzone, so 180 cm advances exploration meaningfully.`

This explanation makes explicit why a particular direction is chosen and shows that exploration is driven by task relevance and environmental structure rather than by uniform or reactive scanning. As a result, movement appears meaningful, and users can infer the rationale behind the drone's behavior.

7.5.6. Shortcuts

Evaluation goal. Shortcuts allow users to assign a custom name to a sequence of actions or behaviors and to recall it later through a single command. The objective of this evaluation is to assess whether this mechanism is perceived as useful by users, how easily it is understood, and in which situations it provides tangible benefits.

The usefulness of shortcuts primarily depends on task complexity, repetition, and the cognitive effort required to repeatedly formulate similar instructions. In particular, shortcuts are expected to be more beneficial when interactions involve structured, multi-step behaviors or recurring action patterns.

Since shortcuts are implemented only in `DACS`, this component is evaluated exclusively within this system.

User-reported metrics. User perception of shortcuts is assessed through post-test questionnaire items asking:

- *whether defining shortcuts was useful during the interaction;*
- *in which scenarios shortcuts would be most beneficial (open-ended).*

Figure 7.7 reports the distribution of responses to the first question.

Responses show heterogeneous evaluations. While several participants rate shortcuts as highly useful (values 5), two participants assign a mid-level score (value 3), indicating uncertainty or limited perceived benefit within the experimental setting. No low scores

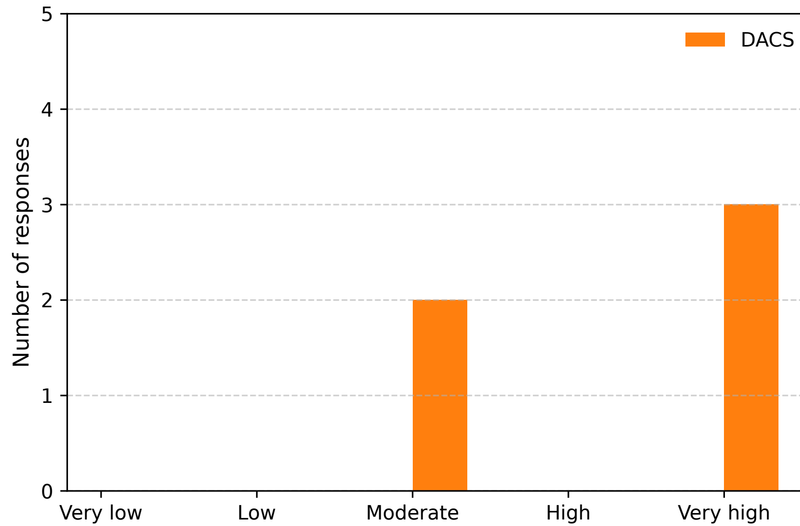


Figure 7.7: User-reported usefulness of defining shortcuts in DACS. Participants rated how beneficial shortcut definition was during interaction.

are observed, suggesting that shortcuts are not perceived as negative, but not universally necessary.

Written answers to the open-ended question provide further insight. Some users explicitly note that shortcuts would become more valuable in more complex or long-term scenarios. For instance, participant U1 writes:

> *"I think in more complex use cases this may be a useful feature."*

Other users report immediate benefits even within the limited indoor environment, particularly when tasks involve complex or expressive behaviors.

Interpretation through interaction logs. Analysis of interaction helps explain the observed user perceptions. Shortcuts are the most demanding adaptive feature to understand and use, as they require users to abstract a sequence of actions into a reusable concept. In simple scenarios with short command sequences, several participants prefer issuing commands directly, perceiving shortcut creation as unnecessary overhead.

However, when interactions involve longer or more structured behaviors, shortcuts significantly reduce cognitive and interaction effort. A representative example is participant U4, who asks the drone to execute a complex sequence of movements resembling a dance. After observing and appreciating the result, the participant saves the behavior as a shortcut named `"dance"` and subsequently re-executes it by writing "*execute the task dance*".

This reduction in mental demand is explicitly reflected in participant comments. Participant U5 writes:

> *"During more casual interactions, being able to issue a single sentence instead of repeatedly explaining the same behavior could significantly reduce effort and make the activity easier."*

The logs also show that the system actively supports users in understanding shortcuts through self-explanation. When users are unsure about the feature, they can directly ask the system for guidance. For example, participant U5 queries:

```
What shortcuts can I use with you?
```

The system responds with a concise tutorial-style explanation, describing how to save, execute, and delete shortcuts, together with usage tips:

```
Shortcuts let you save a whole task and run it later by name.
- Save the last task as a shortcut:  Save last task with shortcut
"morning scan"
- Run a saved shortcut:  Execute task "morning scan"
```

This behavior lowers the barrier to adoption and explains why even users who did not actively rely on shortcuts still report moderate usefulness rather than low scores.

Overall, the results indicate that shortcuts are perceived as a powerful convenience mechanism whose value scales with task complexity and repetition. While not immediately necessary for simple interactions, they become particularly effective for encapsulating complex trajectories, expressive behaviors, or frequently repeated tasks, supporting more concise and cognitively lightweight interaction over time.

7.5.7. User-perceived Latency

Evaluation goal. Latency measures the responsiveness of the system during interaction. The objective of this evaluation is to analyze how delays introduced by the system affect user experience and perceived usability.

Latency is not a primary optimization goal of this work, whose focus lies on interface customization and LLM-enhanced expressiveness. Nevertheless, latency is recorded and analyzed to evaluate its impact on usability and to understand how users perceive waiting times during interaction.

Two sources of latency are considered:

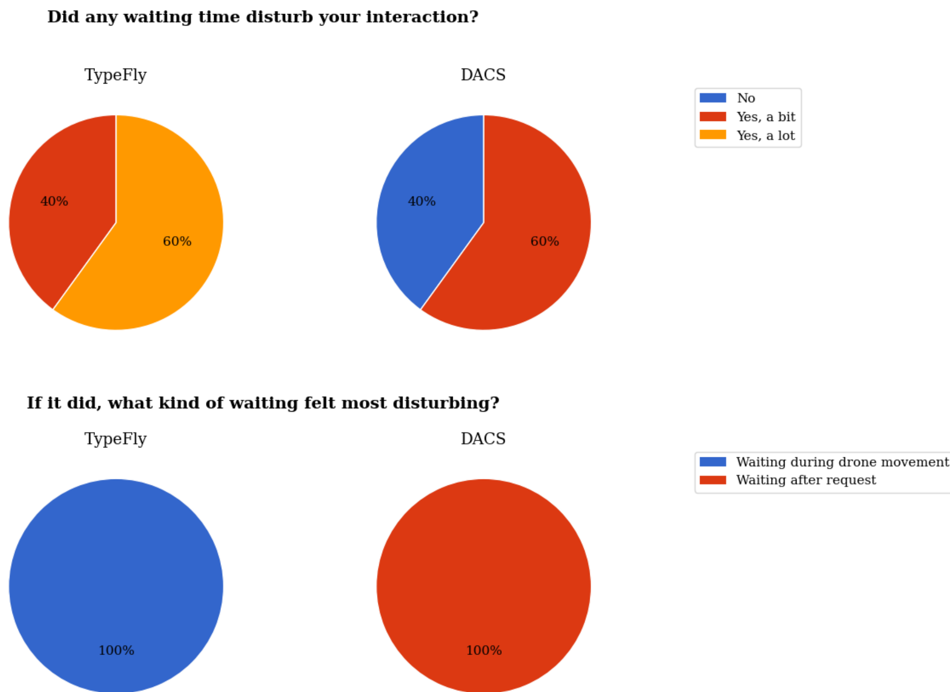


Figure 7.8: User-reported perception of waiting time disturbance and most disturbing waiting phase.

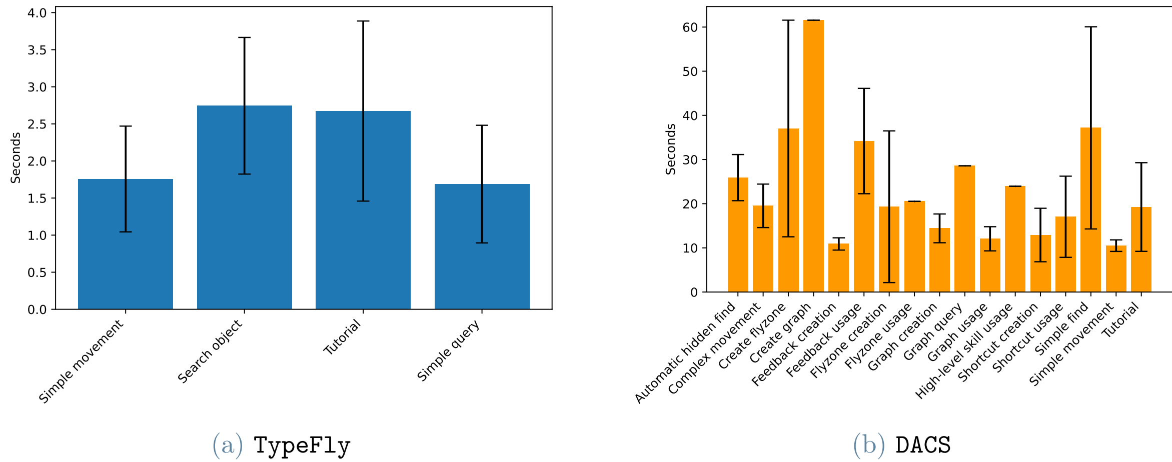
- **planning latency**, defined as the time required to generate a plan after a user request;
- **execution latency**, defined as the time required for the drone to physically execute the planned actions.

The analysis primarily focuses on planning latency, while execution latency is discussed insofar as it influences user perception.

User-reported metrics. User perception of latency is assessed through post-test questionnaire items asking:

- *whether waiting time disturbed the interaction;*
- *which type of waiting was most disturbing (before motion or during drone movement).*

For `TypeFly`, the majority of participants report that waiting time disturbed a lot the interaction. As shown in Figure 7.8, the most disturbing delays are associated with waiting *during drone movement*. This perception reflects situations in which the drone repeatedly scans the environment or rotates in place while attempting to locate non-visible objects, leading to prolonged execution without clear progress, as discussed in Section 7.5.5.



(a) `TypeFly`

(b) `DACS`

Figure 7.9: Mean planning latency (with standard deviation) grouped by request category.

In contrast, Figure 7.8 shows that for `DACS`, waiting times are perceived as less disruptive overall. When delays are reported, they are primarily associated with the time before the drone starts moving, corresponding to planning latency. Notably, users do not express frustration related to execution time. Instead, some participants explicitly suggest that providing immediate feedback indicating that the request is being processed would improve transparency during the planning phase. This behavior could be easily achieved without altering the underlying planning logic, for instance by emitting a short acknowledgment message as soon as the request is received.

Interpretation through interaction logs. To contextualize the questionnaire results, objective latency measurements are extracted from system logs and grouped by category of user request.

Figure 7.9 reports the mean planning latency (with standard deviation) for representative request categories in both systems. As shown, `TypeFly` exhibits consistently low planning latency across all request categories, reflecting its relatively simple and stateless planning pipeline.

In contrast, `DACS` shows higher and more variable planning latency, particularly for requests involving adaptive mechanisms such as context graph usage, feedback integration, flyzone handling, and shortcut execution. This increase is expected, as additional reasoning steps are required to generate context-aware and personalized plans.

Despite this difference, the questionnaire results indicate that increased planning latency in `DACS` does not negatively impact perceived usability. Users primarily associate disruptive waiting with execution-time behavior rather than with plan generation.

Interpretation. The comparison highlights a clear trade-off between planning complexity and execution efficiency. `TypeFly` reacts quickly at the planning stage but often results in inefficient or stalled execution, particularly during search tasks under partial observability. These situations lead to prolonged execution latency and user frustration.

Conversely, `DACS` incurs higher planning latency due to richer reasoning and adaptive mechanisms, but produces more directed and purposeful execution. From a user perspective, short and predictable delays before motion are less disruptive than long or unclear waiting during movement.

Overall, the results indicate that the additional planning latency introduced by `DACS` does not harm usability. Instead, users express a preference for greater transparency during planning, suggesting that interaction quality could be further improved by explicitly signaling that the system is processing a request.

7.5.8. High-Level Skills

Evaluation goal. High-level skills allow the system to encapsulate structured sequences of low-level actions into reusable behavioral abstractions. The objective of this evaluation is to analyze when and why such skills are created by the system, how they are reused across tasks, and how their reuse impacts planning latency.

Unlike other adaptive mechanisms evaluated in this chapter, high-level skills are largely transparent to users. Once introduced, they are invoked implicitly through natural-language interaction and do not require explicit management. For this reason, high-level skills are evaluated exclusively through interpretation of interaction logs rather than through user-reported measures.

Interpretation through interaction logs. Inspection of interaction logs reveals consistent patterns in the conditions under which the `Plan LLM` decides to create a new high-level skill. Skill creation is not triggered by the emergence of reusable behavioral structure.

In particular, new skills are typically introduced when a task:

- requires a structured sequence of actions involving control flow (e.g., loops, conditionals);
- encodes a coherent procedural strategy, such as directional search, autonomous exploration, or fallback behaviors;

- represents a compound navigation routine with clear semantic intent (e.g., returning home, following a boundary);
- incorporates persistent user feedback that modifies how a class of tasks should be executed.

For example, during a directional object search request, the planner synthesizes a skill that scans a constrained angular sector through repeated rotations, image acquisition, and visibility checks. Rather than generating this logic inline, the system encapsulates it as a reusable skill, reflecting the recognition of a general search strategy rather than a one-off plan.

Similarly, complex navigation commands such as returning to a home position and landing, or following the flyzone boundary, are abstracted into dedicated skills that encode multi-step spatial behavior under safety constraints.

A particularly strong trigger for high-level skill creation is the presence of user feedback. When feedback modifies how a task should generally be performed, the planner persistently integrates it by creating or updating a high-level skill.

For instance, when a user specifies that successful object detection should be followed by a celebratory motion, the planner synthesizes a new object-finding skill that embeds the celebratory behavior and subsequently reuses it for future search tasks.

Impact on planning latency. Beyond correctness, high-level skills have a direct and measurable impact on planning latency. Figure 7.10 reports the mean planning latency (with standard deviation) for tasks executed without high-level skills and for equivalent tasks executed using stored skills.

The comparison is computed by selecting only those user requests that the `Plan` LLM identifies as valuable candidates for high-level skill abstraction. For each such request, planning latency is recorded twice. The first measurement corresponds to the initial execution, during which the skill is automatically created and immediately used; this case is effectively equivalent to planning exclusively with low-level actions. The second measurement corresponds to a subsequent execution in which the same behavior is invoked solely through the previously stored high-level skill.

This paired measurement isolates the effect of skill reuse while keeping task semantics constant. The results show a clear reduction in planning latency when a stored skill is available. By reusing an existing abstraction instead of generating a full action sequence from scratch, the planner significantly reduces both average planning time and variability.

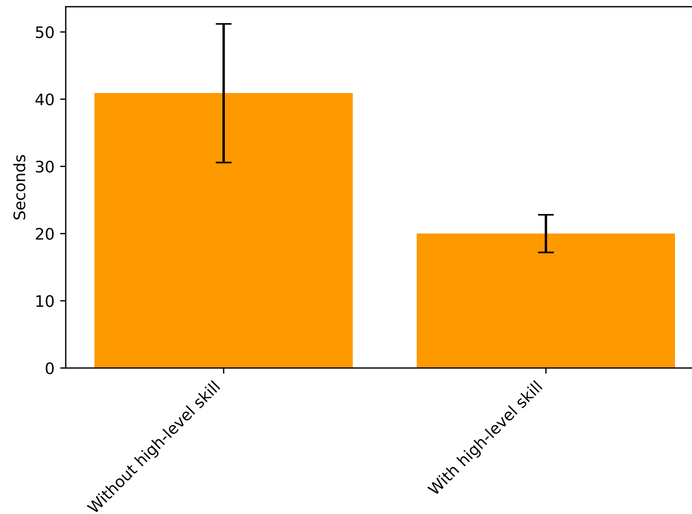


Figure 7.10: Mean planning latency (± standard deviation) for tasks executed without high-level skills and with high-level skills in DACS.

7.6. Summary

This section synthesizes the evaluation findings and provides explicit answers to the research questions defined in Section 7.1.

RQ1 (User adaptation). All five participants interacting with DACS explicitly used at least one customization mechanism during the session, and each issued multiple feedback instructions that were persistently integrated into subsequent tasks. Participants successfully defined flyzones, created shortcuts, and expressed reusable behavioral refinements through natural-language feedback. In contrast, the baseline condition provided no persistent customization mechanisms. Log analysis confirms that user-defined feedback rules were stored and reused across tasks, and that shortcuts were executed as reusable abstractions rather than regenerated action sequences. These results indicate that users are not limited to issuing isolated commands, but can actively modify how the system operates over time, supporting a positive answer to RQ1.

RQ2 (Context awareness). All five participants interacting with DACS reported high perceived memory and contextual understanding, whereas four out of five baseline users reported little or no perception of persistent memory. Interaction logs show that stored object-region associations and spatial constraints were explicitly referenced during exploration and replanning. In hidden-object scenarios, the baseline repeatedly stalled into local scanning, while DACS autonomously selected contextually motivated exploration di-

rections. These findings demonstrate that contextual knowledge is not only accumulated but actively exploited during decision making, providing strong evidence in favor of RQ2.

RQ3 (Overall user experience). Across usability-related measures, `DACS` improves perceived clarity, predictability, and interaction quality compared to `TypeFly`. Users reported lower difficulty in understanding system capabilities and higher confidence in autonomous search behavior. Although planning latency increased due to additional reasoning steps, this did not negatively impact perceived usability. In contrast, disruptive experiences in the baseline were primarily associated with inefficient execution and stalled exploration. Overall, participants described the adaptive system as more structured, transparent, and aligned with their intentions. Within the limits of the study sample, the collected data support a positive answer to RQ3.

Overall interpretation. Taken together, the findings indicate that combining user-driven customization mechanisms with context-aware reasoning produces an interaction paradigm that evolves during use and remains interpretable to users. Rather than functioning as a stateless command executor, `DACS` supports progressively personalized and contextually grounded behavior. While the adaptive architecture introduces additional planning complexity, it yields more purposeful, predictable, and user-aligned interaction, providing coherent support for the proposed framework.

8 Conclusion

This thesis explored the evolution of Human-Drone Interaction toward adaptive and context-aware collaboration, proposing a unified framework that bridges writable interaction paradigms with reasoning-driven autonomy. Starting from the limitations observed in existing systems, particularly the separation between user authorability and cognitive planning, the work introduced `DACS`, a context-aware architecture that integrates natural-language reasoning, environmental memory, and adaptive execution strategies into a continuous interaction loop.

The proposed approach demonstrates how Large Language Models can act not only as planning components but as mediators between perception, memory, and user intent, enabling drones to move beyond static command execution toward progressively evolving behavior. Through mechanisms such as short-term and long-term memory, visual environment awareness, flyzone reasoning, and high-level skill creation, the system transforms interaction into a temporally extended process shaped by context and accumulated experience. As discussed throughout the thesis, this design allows the drone to adapt strategies over time, improving responsiveness and reducing repetitive execution patterns commonly observed in baseline systems.

Results from the comparative user study suggest that embedding memory and contextual reasoning within the interaction loop enhances perceived collaboration, reduces user frustration, and enables more personalized drone behavior compared to traditional reasoning-only approaches.

Despite these promising outcomes, several challenges remain open. The reliance on external reasoning services introduces latency and raises questions regarding robustness in dynamic environments, while the current prototype operates within constrained indoor scenarios. Future work may investigate tighter integration between perception and reasoning, scalable memory representations for long-term deployment, and richer multimodal interaction modalities that further strengthen the sense of shared agency between humans and aerial systems.

Bibliography

[1] P. Abtahi, D. Y. Zhao, J. L. E, and J. A. Landay. Drone near me: Exploring touch-based human-drone interaction. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–8, 2017.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[4] J. Aloimonos, I. Weiss, and A. Bandyopadhyay. Active vision. *International journal of computer vision*, 1(4):333–356, 1988.

[5] O. Alon, S. Rabinovich, C. Fyodorov, and J. R. Cauchard. Drones in firefighting: A user-centered design perspective. In *Proceedings of the 23rd international conference on mobile human-computer interaction*, pages 1–11, 2021.

[6] R. Bajcsy. Active perception. *Proceedings of the IEEE*, 76(8):966–1005, 1988.

[7] C. Bartneck, T. Belpaeme, F. Eyssel, T. Kanda, M. Keijsers, and S. Šabanović. *Human-robot interaction: An introduction*. Cambridge University Press, 2024.

[8] Bitcraze AB. Crazy realtime protocol (crtp). `https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/crtp/`.

[9] Bitcraze AB. Lighthouse positioning system documentation, 2024. URL `https://www.bitcraze.io/documentation/system/positioning/ligthouse-positioning-system/`. Accessed: 2026-02-22.

[10] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*, 2022.

[11] E. Brunswik. *Perception and the representative design of psychological experiments*. Univ of California Press, 2023.

[12] J. Cacace, A. Finzi, V. Lippiello, M. Furci, N. Mimmo, and L. Marconi. A control architecture for multiple drones operated via multimodal interaction in search & rescue mission. In *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 233–239. IEEE, 2016.

[13] F. Caserta. Towards drones as writable surfaces. 2023.

[14] Á. Castro-González, H. Admoni, and B. Scassellati. Effects of form and motion on judgments of social robots' animacy, likability, trustworthiness and unpleasantness. *International Journal of Human-Computer Studies*, 90:27–38, 2016.

[15] J. R. Cauchard, J. L. E, K. Y. Zhai, and J. A. Landay. Drone & me: an exploration into natural human-drone interaction. In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, pages 361–365, 2015.

[16] J. R. Cauchard, M. Khamis, J. Garcia, M. Kljun, and A. M. Brock. Toward a roadmap for human-drone interaction. *Interactions*, 28(2):76–81, 2021.

[17] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), Mar. 2024. ISSN 2157-6904. doi: 10.1145/3641289. URL `https://doi.org/10.1145/3641289`.

[18] G. Charness, U. Gneezy, and M. A. Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1): 1–8, 2012. ISSN 0167-2681. doi: https://doi.org/10.1016/j.jebo.2011.08.009. URL `https://www.sciencedirect.com/science/article/pii/S0167268111002289`.

[19] G. Chen, X. Yu, N. Ling, and L. Zhong. Typefly: Flying drones with large language model. *arXiv preprint arXiv:2312.14950*, 2023.

[20] L. Davies, R. C. Bolam, Y. Vagapov, and A. Anuchin. Review of unmanned aircraft system technologies to enable beyond visual line of sight (bvlos) operations. In *2018 X International conference on electrical power drive systems (ICEPDS)*, pages 1–6. IEEE, 2018.

[21] M. M. de Graaf, S. Ben Allouch, and J. A. Van Dijk. What makes robots social?: A user's perspective on characteristics for social human-robot interaction. In *International Conference on Social Robotics*, pages 184–193. Springer, 2015.

[22] A. D. Dragan and S. S. Srinivasa. A policy-blending formalism for shared control. *The International Journal of Robotics Research*, 32(7):790–805, 2013.

[23] A. D. Dragan, K. C. Lee, and S. S. Srinivasa. Legibility and predictability of robot motion. In *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 301–308. IEEE, 2013.

[24] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, et al. Palm-e: An embodied multimodal language model. 2023.

[25] R. A. S. Fernandez, J. L. Sanchez-Lopez, C. Sampedro, H. Bavle, M. Molina, and P. Campoy. Natural user interfaces for human-drone multi-modal interaction. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1013–1022. IEEE, 2016.

[26] M. Flasiński. Symbolic artificial intelligence. In *Introduction to Artificial Intelligence*, pages 15–22. Springer, 2016.

[27] S. Gallagher. Philosophical conceptions of the self: implications for cognitive science. *Trends in cognitive sciences*, 4(1):14–21, 2000.

[28] W. W. Gaver, J. Beaver, and S. Benford. Ambiguity as a resource for design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, page 233–240, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136307. doi: 10.1145/642611.642653. URL `https://doi.org/10.1145/642611.642653`.

[29] S. Gillies et al. Shapely: Manipulation and analysis of geometric objects, 2007–. URL `https://shapely.readthedocs.io/`. Python library for planar geometric objects based on GEOS.

[30] N. Gio, R. Brisco, and T. Vuletic. Control of a drone with body gestures. *Proceedings of the Design Society*, 1:761–770, 2021.

[31] M. A. Goodrich and A. C. Schultz. *Human-Robot Interaction: A Survey*. Now Publishers Inc., Hanover, MA, USA, 2008. ISBN 1601980922.

[32] V. Herdel, L. J. Yamin, and J. R. Cauchard. Above and beyond: A scoping review of domains and applications for human-drone interaction. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–22, 2022.

[33] K. Hook. *Designing with the body: Somaesthetic interaction design*. MIt Press, 2018.

[34] Imperial War Museum. A brief history of drones. https://www.iwm.org.uk/history/a-brief-history-of-drones, 2024.

[35] S. Javaid, H. Fahim, B. He, and N. Saeed. Large language models for uavs: Current state and pathways to the future. *IEEE Open Journal of Vehicular Technology*, 2024.

[36] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma. A review of yolo algorithm developments. *Procedia Computer Science*, 199:1066–1073, 2022. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2022.01.135. URL https://www.sciencedirect.com/science/article/pii/S1877050922001363. The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 & 2021): Developing Global Digital Economy after COVID-19.

[37] M. N. H. Khan and C. Neustaedter. An exploratory study of the use of drones for assisting firefighters during emergency situations. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–14, 2019.

[38] J. La Delfa, R. Garrett, A. Lampinen, and K. Höök. Articulating mechanical sympathy for somaesthetic human-machine relations. In *Proceedings of the 2024 ACM Designing Interactive Systems Conference*, pages 3336–3353, 2024.

[39] J. La Delfa, R. Garrett, A. Lampinen, and K. Höök. How to train your drone: Exploring the umwelt as a design metaphor for human-drone interaction. In *Proceedings of the 2024 ACM Designing Interactive Systems Conference*, pages 2987–3001, 2024.

[40] J. D. Lee and K. A. See. Trust in automation: Designing for appropriate reliance. *Human factors*, 46(1):50–80, 2004.

[41] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

[42] A. Lykov, M. Konenkov, K. F. Gbagbe, M. Litvinov, D. Davletshin, A. Fedoseev, M. A. Cabrera, R. Peter, and D. Tsetserukou. Cognitiveos: Large multimodal model based system to endow any type of robot with generative ai. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 16256–16261. IEEE, 2025.

[43] A. Lykov, V. Serpiva, M. H. Khan, O. Sautenkov, A. Myshlyaev, G. Tadevosyan, Y. Yaqoot, and D. Tsetserukou. Cognitivedrone: A vla model and evaluation benchmark for real-time cognitive task solving and reasoning in uavs. *arXiv preprint arXiv:2503.01378*, 2025.

[44] J. B. Lyons. Being transparent about transparency. In *AAAI Spring Symposium*, pages 48–53. AAAI Press Palo Alto, California, USA, 2013.

[45] A. C. Medeiros, P. Ratsamee, Y. Uranishi, T. Mashita, and H. Takemura. Human-drone interaction: Using pointing gesture to define a target object. In *International Conference on Human-Computer Interaction*, pages 688–705. Springer, 2020.

[46] K. Natarajan, T.-H. D. Nguyen, and M. Mete. Hand gesture controlled drones: An open source library. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 168–175. IEEE, 2018.

[47] OpenAI. Openai api documentation: Responses api. `https://platform.openai.com/docs/api-reference/responses`.

[48] Z. Ravichandran, V. Murali, M. Tzes, G. J. Pappas, and V. Kumar. Spine: Online semantic planning for missions with incomplete natural language specifications in unstructured environments. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 13714–13721. IEEE, 2025.

[49] T. B. Sheridan. *Telerobotics, automation, and human supervisory control*. MIT press, 1992.

[50] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

[51] M. Sondoqah, F. Ben Abdesslem, K. Popova, M. McGregor, J. La Delfa, R. Garrett, A. Lampinen, L. Mottola, and K. Höök. Shaping and being shaped by drones: Programming in perception-action loops. In *Proceedings of the 2024 ACM Designing Interactive Systems Conference*, pages 2926–2945, 2024.

[52] A. Stoica, F. Salvioli, and C. Flowers. Remote control of quadrotor teams, using hand gestures. In *Proceedings of the 2014 ACM/IEEE international conference on Human-robot interaction*, pages 296–297, 2014.

[53] L. A. Suchman. From interactions to integrations. In *Human-Computer Interaction INTERACT'97: IFIP TC13 International Conference on Human-Computer Interaction, 14th–18th July 1997, Sydney, Australia*, pages 3–3. Springer, 1997.

[54] M. L. Tazir, M. Mancas, and T. Dutoit. From words to flight: Integrating openai chatgpt with px4/gazebo for natural language-based drone control. In *International Workshop on Computer Science and Engineering*, 2023.

[55] G. R. Team, S. Abeyruwan, J. Ainslie, J.-B. Alayrac, M. G. Arenas, T. Armstrong, A. Balakrishna, R. Baruch, M. Bauza, M. Blokzijl, et al. Gemini robotics: Bringing ai into the physical world. *arXiv preprint arXiv:2503.20020*, 2025.

[56] D. Tezza and M. Andujar. The state-of-the-art of human–drone interaction: A survey. *ieee access*, 7:167438–167454, 2019.

[57] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[58] C. Wankmüller, M. Kunovjanek, and S. Mayrgündter. Drones in emergency response–evidence from cross-border, multi-disciplinary usability tests. *International Journal of Disaster Risk Reduction*, 65:102567, 2021.

[59] R. H. Wortham, A. Theodorou, and J. J. Bryson. Robot transparency: Improving understanding of intelligent behaviour for designers and users. In *Conference Towards Autonomous Robotic Systems*, pages 274–289. Springer, 2017.

[60] A. Yeh, P. Ratsamee, K. Kiyokawa, Y. Uranishi, T. Mashita, H. Takemura, M. Fjeld, and M. Obaid. Exploring proxemics for human-drone interaction. HAI '17, page 81–88, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351133. doi: 10.1145/3125739.3125773. URL `https://doi.org/10.1145/3125739.3125773`.

[61] J. E. Young, J. Sung, A. Voida, E. Sharlin, T. Igarashi, H. I. Christensen, and R. E. Grinter. Evaluating human-robot interaction: Focusing on the holistic interaction experience. *International Journal of Social Robotics*, 3(1):53–67, 2011.

[62] Z. Zhu, J.-Y. Cheng, I. Jeelani, and M. Gheisari. Using gesture and speech communication modalities for safe human-drone interaction in construction. *Advanced Engineering Informatics*, 62:102827, 2024.

[63] B. Zitkovich, T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR, 2023.

A | Questionnaires

This appendix reports pre-test and post-test questionnaires administered before the experimental session.

Pre-Test Questionnaire

1) **What is your age?**

2) **What is your gender?**

◯ Male ◯ Female ◯ Other: ....

3) **What is your highest level of education?**

◯ Middle school
◯ High school
◯ Bachelor's degree
◯ Master's degree
◯ PhD
◯ Other: ....

4) **Have you ever controlled a drone before?**

◯ Yes, regularly
◯ Yes, a few times
◯ No, never

5) **How familiar are you with vocal or natural-language interfaces (e.g., Alexa, Siri, Google Assistant, etc.)?**

(1 – Not familiar at all, 5 – Very familiar)

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

6) **How familiar are you with AI assistants (e.g., ChatGPT, Siri, etc.)?**
(1 – Not familiar at all, 5 – Very familiar)
○ 1 ○ 2 ○ 3 ○ 4 ○ 5

7) **How familiar are you with new interactive technologies?**
(1 – Not familiar at all, 5 – Very familiar)
○ 1 ○ 2 ○ 3 ○ 4 ○ 5

8) **What do you expect a natural-language-controlled drone to be capable of?**

9) **What do you expect might be difficult?**

10) **How intuitive do you think interacting with a natural-language drone will be?**
(1 – Not intuitive at all, 5 – Very intuitive)
○ 1 ○ 2 ○ 3 ○ 4 ○ 5

11) **How autonomous do you expect the drone you will try to be?**
(1 – Not autonomous at all, 5 – Fully autonomous)
○ 1 ○ 2 ○ 3 ○ 4 ○ 5

12) **Are you concerned or curious about anything before starting?**

Post-Test Questionnaire

1) **How intuitive was it to interact with the system?**
(1 – Not intuitive at all, 5 – Very intuitive)
○ 1 ○ 2 ○ 3 ○ 4 ○ 5

2) **How easy was it to understand what the drone was doing at any moment?**
(1 – Not clear at all, 5 – Very clear)
○ 1 ○ 2 ○ 3 ○ 4 ○ 5

3) **How confident did you feel while giving instructions?**

(1 – Not confident at all, 5 – Very confident)

○ 1 ○ 2 ○ 3 ○ 4 ○ 5

4) **Overall, how satisfied are you with the interaction?**
   (1 – Not satisfied at all, 5 – Very satisfied)

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

5) **How clear were the scenarios you were asked to perform?**
   (1 – Not clear at all, 5 – Very clear)

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

6) **How difficult was it to understand what the system was capable of while using it?**
   (1 – Not difficult at all, 5 – Very difficult)

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

7) **How much mental effort did the interaction require?**
   (1 – Very low effort, 5 – Very high effort)

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

8) **If you found the scenarios mentally demanding, which aspects contributed the most?**

9) **How would you rate your own performance in the scenarios?**
   (1 – Very unsuccessful, 5 – Very successful)

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

10) **If you felt you struggled or could not complete some parts, what challenges did you encounter?**

11) **How fast did the drone feel when reacting to your requests?**
    (1 – Very slow, 5 – Very fast)

    ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

12) **Did any waiting time disturb your interaction?**

◯ No ◯ Yes, a bit ◯ Yes, a lot

13) **If it did, what kind of waiting felt most disturbing?**

◯ Waiting after I sent my request, before drone movements
◯ Waiting during drone movements
◯ Other: ....

14) **Did the drone seem to remember previous regions, objects, or context?**

(1 – Not at all, 5 – Very clearly)

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

15) **During the session, did the system's behavior feel more like:**

◯ Starting from zero every time
◯ Reusing what happened earlier in the session
◯ Not sure

16) **How natural did the system's behavior feel?**

(1 – Very unnatural, 5 – Very natural)

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

17) **When you corrected or refined your request, how much did the system's behavior change?**

(1 – Did not improve, 5 – Improved a lot)

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

18) **How useful was that change? (Leave empty if not available)**

(1 – Not useful, 5 – Very useful)

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

19) **Briefly describe that moment if available.**

20) **Have you found it useful to define a shortcut?**

(1 – Not useful at all, 5 – Very useful)

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

21) **In which scenarios do you think shortcuts would be most useful or beneficial?**

22) **How comfortable did you feel with where the drone moved during the session?**
(1 – Not comfortable, 5 – Very comfortable)
◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

23) **Did the drone tend to stay in the areas where you expected it to be?**
(1 – Not at all, 5 – Completely)
◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

24) **When the drone had to "look around" or explore, how reasonable did its choices feel?**
(1 – Not reasonable at all, 5 – Very reasonable)
◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

25) **When the target was not immediately visible, how confident were you that the drone was searching appropriately?**
(1 – Not confident at all, 5 – Very confident)
◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

26) **What was the most satisfying part of using this system?**

27) **What was the most frustrating part?**

28) **If you could change one thing in this system, what would it be?**

29) **Any other comments or thoughts you would like to share?**

List of Figures

1.1 Evolution of Human-Drone Interaction paradigms. 2
1.2 Conceptual model of DACS. The LLM acts as the central reasoning component, mediating interaction between the user, memory, environment, and drone. 4

3.1 Overview of TypeFly [19]. (1) The user provides a natural-language task description. On the edge server, the Prompt Generator combines the task with a scene description produced by the Vision Encoder and with the available system skills. (2-3) The resulting prompt is sent to a cloud-based LLM, which returns a structured execution plan encoded in MiniSpec. (4) The plan is executed by the MiniSpec Runtime, which may trigger replanning or probing based on runtime feedback (6). (5) Control commands are sent to the drone's motion controller, while visual input is continuously captured to update the scene description. 30

4.1 Conceptual model of DACS. The LLM acts as the central reasoning component, interacting with the user, memory, environment, and the drone, which executes the generated plans. 46
4.2 Dual-prompt interaction model in DACS. Each LLM instance is configured with a fixed system prompt that defines its role, while user instructions are provided separately through user prompts. 48
4.3 *Multi-level memory* architecture of DACS. 52
4.4 Example flyzone composed of five axis-aligned polygons generated by the system. The two large square polygons represent the main rooms (500 cm × 500 cm), connected by a central corridor polygon of dimensions 380 cm × 100 cm. Two thin black vertical polygons located at the corridor entrances denote transition boundaries between the rooms and the corridor. Colors indicate distinct polygonal regions. Together, these polygons define the admissible spatial domain in which the drone is allowed to operate. 60

5.1 Layered software architecture of DACS. 65

5.2 Overview of the multi-LLM interaction pattern in `DACS`. The `Plan LLM` acts as the central orchestrator and triggers auxiliary LLM instances through dedicated low-level skills during plan execution. All auxiliary instances are invoked on demand, except for the `Short-Term Memory` instance, which is activated only when a task requires a new planning iteration based on the outcomes of previous executions. 75

5.3 Example input output interaction of the `Query LLM` instance in `DACS`. The input includes the current camera image, the list of objects detected by the YOLO based Vision Encoder with their bounding boxes, and a natural language query such as *"How many apples are there in the scene"*. By jointly reasoning over the symbolic detections and the visual content of the image, the `Query LLM` instance produces a concise answer, in this case an integer count of the apples visible in the scene. 78

5.4 Sequence diagram for task-related feedback persistence and reuse. (a) The user issues a task and the `Plan LLM` generates an execution plan that is carried out by the drone. (b) After execution, the user provides task-related feedback, which triggers the `Save Task Feedback LLM` and results in a structured summary stored as long-term memory. (c) During a subsequent task, previously stored feedback is retrieved and injected into the planning context, allowing the `Plan LLM` to generate an adapted MiniSpec plan that reflects prior user preferences. 80

5.5 External view of the evaluation environment during a *find an apple* task. The yellow apple is occluded behind a vertical structure, while a banana is clearly visible in the scene. In this configuration, the target object is not directly detectable from the current viewpoint, requiring exploration beyond immediate perception. Based on visual context, execution history, semantic associations, and spatial constraints, the `Choose Direction LLM` generates a structured navigation decision (e.g., `yaw = 0`, `distance = 150 cm`, `region_name = "office"`), reasoning that areas containing food-related objects represent semantically promising exploration directions. 83

5.6 Initializing environment knowledge from a real user-provided description: *"I want to describe the environment where you will execute the plans. The area is composed of two circular regions. The first circle is centered at (0, 0) with radius 50 cm and is named christian_area. The second circle is centered at (0, 150) with radius 50 cm and is named luca_area. The two circles are connected by a corridor of width 50 cm."* 85

6.1 Hardware platform of DACS. The DJI Tello quadrotor provides actuation and visual sensing, while a Bitcraze Crazyflie 2.1 equipped with a Lighthouse positioning deck is rigidly mounted on top of the drone and supplies accurate indoor localization. 90

7.1 Indoor office environment used during the evaluation. 97
7.2 User-reported difficulty in understanding system capabilities. Responses are reported for both TypeFly and DACS. 103
7.3 User-reported feedback effectiveness for TypeFly and DACS. Each subplot corresponds to one questionnaire item, with colors distinguishing the two systems. 106
7.4 User-reported perception of memory and contextual understanding for the baseline TypeFly and DACS. Participants rated whether the drone appeared to remember previously visited regions, objects, or contextual information. 109
7.5 User-perceived spatial safety and predictability across systems. Participants rated whether the drone tended to stay within the areas they expected it to be. 111
7.6 User-reported confidence in directional search behavior when the target is not immediately visible. The merged diagram compares the baseline TypeFly with DACS. 114
7.7 User-reported usefulness of defining shortcuts in DACS. Participants rated how beneficial shortcut definition was during interaction. 116
7.8 User-reported perception of waiting time disturbance and most disturbing waiting phase. 118
7.9 Mean planning latency (with standard deviation) grouped by request category. 119
7.10 Mean planning latency ($\pm$ standard deviation) for tasks executed without high-level skills and with high-level skills in DACS. 122

List of Tables

2.1 Summary of HDI paradigms and emerging directions. 27

5.1 Summary of verbosity and reasoning effort configuration for the LLM instances in `DACS`. 74

7.1 Pre-test profiles of participants in both conditions. *Tech* indicates a technical or engineering background. *AI* denotes familiarity with Artificial Intelligence systems, and *NL* denotes familiarity with Natural Language interfaces, both measured on a 1-5 Likert scale (1 = none, 5 = very high). 96