**THE FOLLOWING DOCUMENT WAS ASSITED WITH LLM**

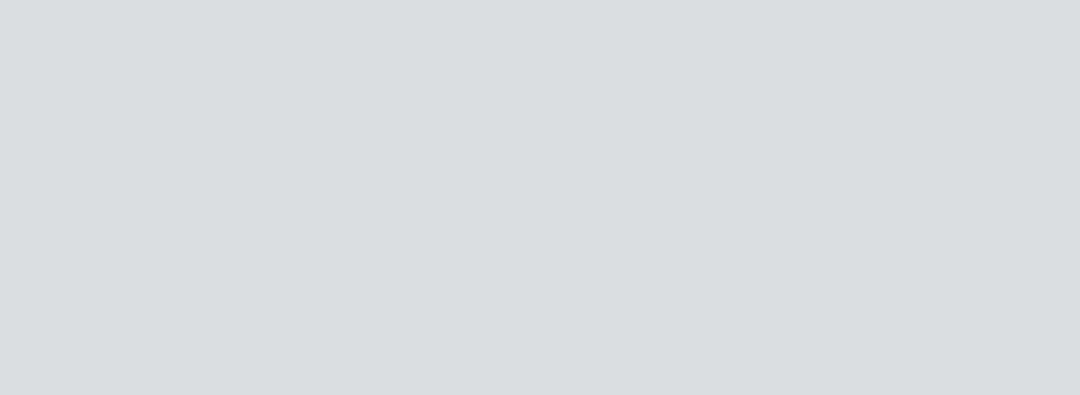
UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI

FACULTATEA DE AUTOMATICĂȘI CALCULATOARE

DEPARTAMENTUL DE CALCULATOARE

A circular logo with a building in the center

AI-generated content may be incorrect.



Computer Science & Engineering Depart ment

PROIECT DE DIPLOMĂ

Orchestrarea robotizată a sarcinilor casnice în cadrul RoboCasa utilizând arbori comportamentali și Învățarea prin recompensă

Luca Plian

**Coordonator stințific:**

### Prof. dr. ing. Alexandru Sorici

**BUCUREȘTI**

UNIVERSITY POLITEHNICA OF BUCHAREST

FACULTY OF AUTOMATIC CONTROL AND COMPUTERS

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

|  |  |
| --- | --- |
| upb | cs |

DIPLOMA PROJECT

Robotic Orchestration of Household Tasks in the RoboCasa Framework Using Behavior Trees and Reinforcement Learning

Luca Plian

**Thesis advisor:**

### Prof. dr. ing. Alexandru Sorici

**BUCHAREST**

**CUPRINS**

[Sinopsis 2](#_iv4mrvat8q6p)

[Abstract 2](#_p3w0bgu19nh8)

[Mulțumiri 3](#_hx96q2cuntcf)

[1 Introducere 4](#_jiqf87jf8ekx)

[1.1 Context 4](#_512jvur0mi8x)

[1.2 Problema 4](#_yusnw8pe9du0)

[1.3 Obiective 4](#_6xkl886xaeye)

[1.4 Structura lucrării 4](#_g64wj9nh479j)

[2 Analiza și specificarea cerințelor 5](#_jk2tdymxlsay)

[3 Studiu](#_i0elxf57mxi5)

[de piață / Abordări existente 6](#_i0elxf57mxi5)

[3.1 Indicații formatare figuri 6](#_okhvt9kgzsbd)

[4 Soluția propusă 8](#_4pmv5i96s36j)

[4.1 Indicații formatare formule 8](#_k32uyk5yo34m)

[5 Detalii de implementare 9](#_1alv1o8rv90)

[5.1 Indicații formatare tabele 9](#_rwifa1stz1g8)

[6 Studiu de caz / Evaluarea rezultatelor 11](#_d0r2ikfep3p0)

[7 Concluzii 13](#_w7c2okqwfidf)

[8 Bibliografie 14](#_7ajjenejjvlq)

[9 Anexe 16](#_zxdh7p5a7v0)

**Sinopsis**

Sinopsisul proiectului are rol de introducere, conținând atât o descriere pe scurt a problemei abordate cât și o enumerare sumară a rezultatelor și a concluziilor. Se recomandă ca sinopsisul să fie redactat într-un limbaj accesibil unei persoane nefamiliarizate cu domeniul, dar în același timp destul de specific pentru a oferi rapid o vedere de ansamblu asupra proiectului prezentat.

Sinopsisul proiectului va fi redactat atât în română cât și în engleză. Ca dimensiunea recomandată aceasta secțiune va avea maxim 200 de cuvinte pentru fiecare variantă. Împreună, ambele variante se vor încadra într-o singură pagină.

Nasiriany et al. (2024) arată că simulatoarele avansate, precum RoboCasa, sunt esențiale pentru a învăța roboții să îndeplinească sarcini cotidiene. Pe măsură ce tehnologiile pentru case inteligente și pentru asistența la domiciliu (AAL) devin din ce în ce mai răspândite, crește nevoia de roboți care să poată ajuta în mod activ în gospodării, îndeplinind sarcini fizice.

Acest proiect prezintă o metodă care permite unui robot să înțeleagă și să execute activități casnice complexe. Abordarea se bazează pe arbori comportamentali (BT), care servesc ca structură logică ce ghidează robotul pas cu pas printr-o secvență de acțiuni. Spre deosebire de scripturile simple bazate pe reguli, BT permit robotului să se adapteze dinamic atunci când ceva nu merge conform planului, repetând acțiunile sau alegând strategii alternative.

Metoda propusă a fost evaluată în mediul de simulare RoboCasa, unde un robot a fost învățat să „fiarbă două ouă”. Experimentele au demonstrat că robotul putea descompune o sarcină complexă în acțiuni mai mici și putea lua decizii inteligente pentru a finaliza procesul, chiar și în condiții neașteptate.

Lucrările viitoare explorează integrarea învățării prin întărire (RL) pentru a permite robotului să învețe din experiență și să-și îmbunătățească performanța în timp. Acest lucru contribuie la progresul roboților domestici autonomi capabili să facă viața de zi cu zi mai ușoară și mai sigură.

# Abstract

The abstract has an introductory role and should engulf both a brief description of the issue at hand, as well as an overview of the obtained results and conclusions. The abstract should be formulated such that even somebody that is unfamiliar with the projects’ domain can grasp the objectives of the thesis while, at the same time, retaining a specificity level offering a bird’s eye view of the project.

The projects’ abstract will be elaborated in both Romanian and English. The recommended size for this section is limited to 200 words for each version. Together, both versions will fit in one page.

-short description of the approached problem

-short enumeration of the results and conclusions

-a short presentation about the project

(maximum 200 words)

Nasiriany et al. (2024) show that advanced simulators such as **RoboCasa** are essential for teaching robots to perform everyday tasks. As smart home and **Ambient Assisted Living (AAL)** technologies become more widespread, there is a growing need for robots that can actively assist within households by carrying out physical tasks.

This project presents a method that enables a robot to **understand and execute complex household activities**. The approach is based on **Behavior Trees (BTs)**, which serve as a logical structure that guides the robot step by step through a sequence of actions. Unlike simple rule‑based scripts, BTs allow the robot to **adapt dynamically** when something does not go according to plan—retrying actions or choosing alternative strategies.

The proposed method was evaluated in the **RoboCasa** simulation environment, where a robot was taught to “boil two eggs.” Experiments demonstrated that the robot could decompose a complex task into smaller actions and make intelligent decisions to complete the process, even under unexpected conditions.

Future work explores the integration of **Reinforcement Learning (RL)** to enable the robot to learn from experience and improve performance over time. This contributes to the advancement of **autonomous domestic robots** capable of making everyday life easier and safer.

# Mulțumiri

(opțional) Aici puteți introduce o secțiunea specială de mulțumiri / acknowledgments.

# Introducere

Parametrii de formatare recomandați pentru lucrare:

● Font recomandat: Calibri; Dimensiune font: 12;

● Spațiere între linii: 1,15; Spațiere după paragraf: 8pt;

● Stil: Justified;

● Dimensiune pagină: A4; Margini: 2,54cm/ 2,54cm/ 2,54cm/ 2,54cm;

● Heading1: Calibri, 14, bold, all caps;

● Heading2: Calibri, 14, bold;

● Heading3: Calibri, 12.

1. Font pentru formule: Cambria Math, 12.  
     
   În cadrul introducerii, este necesară abordarea următoarelor puncte care reprezintă de fapt familiarizarea cititorului (comisia, alți colegi sau experți în domeniu) cu tema proiectului, soluția propusa și cuprinsul/structura lucrării. Deși introducerea poate conține și unele elemente mai generale, se recomandă păstrarea unui limbaj tehnic, specific audienței care va citi lucrarea.  
     
   În cadrul capitolelor următoare, veți regăsi o serie notații de forma [Dezvoltare de produs], [Cercetare]. Acest tip de formatare este utilizat exclusiv în acest template pentru a marca sfaturi și cerințe specifice pentru lucrări de diploma cu specific diferit. În pregătirea documentului vostru, nu veți utiliza aceste marcaje.  
     
   Elementele pe care trebuie să le abordați în introducere sunt descrise în cadrul subcapitolelor de mai jos.

## 1.1 Context

O scurtă introducere a proiectului, motivație, explicație de ce este relevant domeniul proiectului.

With the continuous advancement and increasing integration of technology in everyday life, there has been significant growth in the adoption of **smart homes** and **IoT (Internet of Things)** devices. This trend is strongly supported by data from Statista, which predicts that by **2030**, the number of connected IoT devices will reach **31.2 billion**, up from **19.8 billion** in 2025 [4].

Within the smart home ecosystem, a specialized subfield known as **Ambient Assisted Living (AAL)** has emerged. AAL systems are primarily designed to support vulnerable populations—such as elderly individuals or people with chronic conditions like diabetes—by improving safety, comfort, and independence through technology.

However, there is an increasing need to extend these intelligent systems to the **general population**, which also spends significant time and effort performing daily household tasks. According to **EIGE**[5], men spend on average **1.6 hours** per day on housework, while women spend about **2.3 hours**. Reducing this time would allow individuals to engage in more meaningful, productive, and fulfilling activities.

The main motivation behind this project is the **development of a robust software architecture for domestic task automation**, aiming to transition from **simple scripted automations** toward **intelligent robotic orchestration**. Unlike traditional smart home setups, the focus here is not only on passive perception but on **active execution**—enabling autonomous robots to plan, act, and recover from errors in realistic household environments.

To achieve this, the **RoboCasa** robotic simulator is used as a testbed for developing and validating an autonomous agent capable of **planning, executing, and managing failures** during complex domestic tasks. This provides a controlled yet realistic environment where different approaches to robotic decision-making, such as **Behavior Trees** and **Reinforcement Learning**, can be explored effectively.

## 1.2 Problema

Care este problema pe care proiectul o va rezolva.

The main problem addressed in this project is the **lack of intelligent, adaptive automation** capable of actively supporting users in their daily household routines.  
While current smart home and IoT systems can monitor user behavior or trigger pre-set actions, they remain limited to **simple, linear automations** (e.g., IFTTT-style scripts). These approaches are **fragile**, unable to manage unexpected errors (such as a failed device command or missing input), and lack the ability to **coordinate complex multi-step activities**.

Existing **Ambient Assisted Living (AAL)** systems mostly focus on vulnerable groups such as the elderly or people with chronic conditions, often overlooking the **general population**, whose daily routines are also burdened by time-consuming domestic tasks.  
According to **Eurostat**  [5], household and family care (excluding childcare) occupies between **35 minutes and 1 hour 17 minutes** per day in EU countries. Automating these activities could not only save time but also **reduce cognitive and mental load**, allowing individuals to focus on more meaningful, productive, or creative pursuits.

A further limitation is that most existing automation systems rely on **static perception and rigid environments**. They are unable to identify or interact with objects that are **partially occluded or placed in dynamic locations** (e.g., inside a fridge or cabinet) and cannot adapt to environmental changes. Moreover, they typically lack robust, high-level decision-making architectures that enable **autonomous adaptation and recovery**.

This project addresses these gaps by exploring **Behavior Trees (BTs)** for orchestrating complex household tasks in a **hierarchical and fault-tolerant** manner, complemented by **Reinforcement Learning (RL)** to improve adaptability and robustness in dynamically changing domestic environments.

## 1.3 Objective

Care sunt obiectivele proiectului/soluției/abordării/ideii; Ce creșteri sau evoluții determină rezolvarea proiectului.

The objective of this project is to **design, implement, and validate a robust software architecture** for autonomous household task execution in realistic, dynamic environments.  
The architecture combines **symbolic task orchestration** using Behavior Trees (BTs) with **adaptive learning mechanisms**based on Reinforcement Learning (RL) to enable a robot to perceive, plan, and act intelligently within the **RoboCasa**simulation framework.

The specific objectives are:

1. **To define** a complex, multi-stage household task — “Boil Two Eggs” — as a **formal procedural plan**(boil\_two\_eggs.yaml), including all required subtasks, object interactions, and success conditions.
2. **To design** a **Behavior Tree–based decision framework** that coordinates and monitors atomic robot actions, ensuring robust execution with built-in **Fallback** and **Retry** mechanisms for error recovery.
3. **To implement** perception and reasoning modules that allow the robot to **identify and locate objects**, even when they are **occluded** (e.g., inside a fridge or cabinet), and to **adapt its trajectory** according to updated object coordinates within the **RoboCasa** environment.
4. **To integrate** a **Reinforcement Learning (RL)** component that enables the robot to **adapt autonomously to environmental changes** (e.g., varying object positions or workspace layouts), continuously improving its task efficiency and decision-making over time.
5. **To validate** the overall system using an **“Oracle-first”** approach, testing the complete planning-to-execution loop in simulated dynamic scenarios and verifying the architecture’s robustness and adaptability.

Through these objectives, the project aims to demonstrate a **hybrid intelligent control system** that merges **high-level symbolic reasoning** with **data-driven adaptive learning**, paving the way toward truly autonomous domestic robots.

## 1.4 Soluția propusă

Descrierea pe scurt a soluției implementate; ce abordare este propusă (nu detalierea utilitarelor și a tehnologiilor, ci abordarea și ideea propusă de către autor).

- i will describe the conceptual model right now

(generated by GEMINI)

**The proposed solution is a robust, multi-layered software architecture designed for the autonomous execution of complex, multi-stage household tasks within the RoboCasa simulator.**

**The core idea is to decouple the *task definition* (the "what") from the *execution logic* (the "how"). This is achieved through three main components:**

1. **A Formal Task Plan (boil\_two\_eggs.yaml):** The "what." The entire task is first defined as a high-level, human-readable procedural plan1. This plan specifies the preconditions, the objects involved, and the logical sequence of steps required to complete the task 2.
2. **A Behavior Tree (BT) Orchestrator (BT\_complex.mmd):** The "how." This is the "brain" of the solution3. Instead of a simple, brittle script, a Behavior Tree is used to orchestrate the plan. The BT translates the simple sequence from the YAML file into a robust execution flow that can explicitly model and manage real-world complexities, such as failure (Fallback/Abort) and transient errors (Retry logic) 444444444.
3. **An "Oracle-First" Implementation Strategy:** To validate the architecture, the solution is implemented using an "Oracle-first" approach5. The atomic actions of the Behavior Tree ("leaves") are implemented as Python functions that directly call the RoboCasa simulator's API6666. The success or failure of these actions is initially determined not by a complex perception system (like YOLO), but by querying the simulator's true state (the "Oracle") 7777.

This approach allows for the complete validation of the task logic (Plan -> BT -> API calls) independently, proving the robustness of the execution engine before adding the complexity of visual perception.

## Rezultatele obținute

Descriere pe scurt a rezultatelor obținute, eventual de ce acestea sunt importante față de alte soluții sau studii

**1.6** **Structura lucrării**

Un paragraf în care fiecare dintre secțiunile următoare este prezentată în 1-2 fraze, punând accentul pe elementele cele mai semnificative din fiecare secțiune.

Chapter 1 there is a short introduction into the diploma project and its background, problems addressed, objectives and proposed solution.

Chapter 2 presents the existing market demands by surverying people and presents existing research papers and technologies that were previously used within the domain of Ambient Assisted Living that have some sort of connection to my work.

Chapters 3, 4 present the proposed implementation to solve the problem above, Chapter 3 presenting the conceptual work part and Chapter 4 presents the work in more detail.

# Analiza și specificarea cerințelor

[Dezvoltare de produs] Acest capitol va analiza cerințele produsului din prisma potențialilor clienți și a scenariilor de utilizare preconizate, urmând a fi generată o lista de funcționalități.

[Cercetare] Acest capitol va introduce motivația realizării proiectului propus.

Dacă proiectul de licență face parte dintr-un proiect mai amplu (de exemplu un proiect complex, la care lucrează 2 studenți (ex: 1 student la front-end ul aplicației, 1 student la backend-ul aplicației), în acest capitol va fi explicat pe scurt ansamblul proiectului și ce parte din proiect este adresată de lucrarea propusă.

Criterii pentru calificativul Nesatisfăcător:

* [Dezvoltare de produs] Cerințele sunt imaginate de student pe baza unei analize a pieței;
* [Cercetare] Nu se oferă o motivație validă.

Criterii pentru calificativul *Satisfăcător*:

* [Dezvoltare de produs] Există un interviu, un client, analiza cerințelor este elaborată pe baza interviului;
* [Cercetare] Motivația este doar personală.

Criterii pentru calificativul *Bine*:

* [Dezvoltare de produs] Proces iterativ pe baza unor interviuri cu mai mulți clienți, dezvoltare MVP, reevaluare cerințe;
* [Cercetare] Motivația este legată de o necesitate științifică / tehnică explicită.

This chapter provides a detailed analysis of the functional and non‑functional requirements of an **autonomous robotic system** capable of executing complex, multi‑stage domestic tasks in dynamic home environments.  
The proposed system moves beyond simple trigger‑based automation (e.g., IFTTT‑style logic) by enabling **robust, sequential orchestration** and **adaptive error‑handling mechanisms** within a simulated smart‑home context.

Rather than relying on hypothetical user interviews, the design process is driven by a **representative scenario—“Boil Two Eggs”**—implemented in the **RoboCasa** simulator.  
This scenario exposes critical research and engineering challenges, including **multi‑location navigation** (fridge, counter, sink, stove), **object manipulation**, **dynamic environment sensing**, and **fail‑safe task management**.

The project forms part of a broader research initiative on adaptive robotic assistants.  
The present thesis specifically addresses the **Task Execution and Orchestration Module**, which bridges high‑level planning and low‑level control through the **RoboCasa API**, enabling the robot to execute complex procedural tasks autonomously while reasoning about success, failure, and retry strategies.

### **2.1  Motivation and Scope**

Modern IoT and AAL systems provide only limited automation, relying mainly on fixed sequences and simple event‑triggered rules. These lack adaptability, perception under occlusion, and robust recovery mechanisms.  
The proposed system aims to overcome these limitations through a **hybrid approach** combining:

- **Symbolic decision‑making** using **Behavior Trees (BTs)** for deterministic task sequencing;  
- **Perception and reasoning modules** to detect and localize objects even when hidden or misplaced (e.g., inside a refrigerator);  
- **Reinforcement Learning (RL)** algorithms that allow continuous improvement by adapting to environmental changes.

This combination ensures that the robot can **plan, act, and recover** autonomously based on real‑time feedback obtained through the RoboCasa API and simulated sensory data.

### **2.2  Use‑Case Scenarios**

- sistemul intereactiune dintre utilizator si mediul lor casnic

- sa prezint ce fac cu dispotizitele

- doua scenarii in care datele colectate sunt procesate pt a detecta comportamente

- declansa actiuni si oferi feedback

- scenarii functionala si non-functionale

The following use cases describe typical interactions between the **autonomous robot** and the **RoboCasa simulation environment**.  
Each scenario illustrates expected behavior under different operating conditions, validating the system’s functional and non‑functional requirements, including orchestration logic, API integration, and error recovery.

#### **Use‑Case 1 – Nominal Execution (“Happy Path”)**

**Purpose:** Demonstrate successful, uninterrupted completion of a multi‑stage household activity (Boil Two Eggs).

**Actors:** Behavior Tree (BT) Runner; RoboCasa API; Oracle; Environment Simulator.

**Preconditions:**

* boil\_two\_eggs.yaml plan correctly loaded.
* Pot located in cabinet; both eggs present in fridge; appliances (sink, stove) active in the simulator.

**Main Flow:**  
1. BT Runner initializes and queries the Oracle for environment state.  
2. All preconditions are verified; nodes return **SUCCESS**.  
3. Sequential execution begins — each leaf node (e.g., PickPlaceCabinetToCounter, FillPot, BoilWater) sends action commands through the **RoboCasa API**.  
4. Each action validates its result via Oracle queries (e.g., object\_utils.check\_obj\_fixture\_contact).  
5. Upon completion, the BT root node reports **SUCCESS**, confirming task completion.

**Postconditions:**

* Pot with two boiled eggs is successfully placed on the counter.
* Execution logs confirm correctness via the Oracle.

#### **Use‑Case 2 – Critical Failure (Fallback Logic)**

**Purpose:** Validate safe handling of a non‑recoverable failure.

**Actors:** BT Runner; RoboCasa API; Oracle; User Notification Module.

**Preconditions:** One egg missing in the fridge.

**Main Flow:**  
1. BT Runner begins execution; queries Oracle for initial conditions.  
2. Check eggs ≥ 2? node fails validation (returns **FAILED**).  
3. BT logic triggers the **Fallback** branch, executing NotifyUser(error\_message) and halting the plan.

**Alternate Flow:**

* (Future extension) If an RL‑based reasoning module is present, it may attempt to adapt by locating an alternative ingredient or updating the plan.

**Postconditions:**

* System terminates safely with **FAILED** status; no unsafe actions performed.
* User is notified through the interface or simulator log.

#### **Use‑Case 3 – Transient Failure (Retry Logic)**

**Purpose:** Demonstrate robust handling of a recoverable execution error.

**Actors:** BT Runner; RoboCasa API; Oracle.

**Preconditions:** Valid environment state (all objects available).

**Main Flow:**  
1. BT Runner initiates standard sequence.  
2. PickPlaceCabinetToCounter(egg\_2) node fails on first attempt due to an API‑reported grasp error.  
3. The **Retry** decorator captures the failure, increments a retry counter, and re‑executes the same action.  
4. Action succeeds on the second attempt; leaf returns **SUCCESS** and flow continues.

**Postconditions:**

* Task continues seamlessly and completes successfully.
* Retry statistics logged for evaluation of robustness metrics.

#### **Use‑Case 4 – Adaptive Learning Scenario (Reinforcement Learning Extension)**

**Purpose:** Showcase future integration of learning capabilities for perception and decision improvement.

**Actors:** BT Runner; RL Agent; RoboCasa API; Oracle.

**Preconditions:** Dynamic environment — object positions altered (e.g., pot moved, fridge door state changed).

**Main Flow:**  
1. BT initiates the procedural plan.  
2. When perception detects mismatch (object not at stored coordinates), the **RL agent** updates its navigation or search policy based on previous rewards.  
3. The system recalculates target coordinates using Oracle feedback and resumes execution with the updated policy.

**Postconditions:**

* Robot successfully adapts to layout changes and completes task autonomously.
* RL module’s reward function updated for future optimizatio

**2.3 Functional and non-functional requirements**

**Functional (what the system must do):**

* **FR‑1  — Plan Parsing:**  
  The system must interpret a formal procedural plan (boil\_two\_eggs.yaml) to extract its hierarchy, parameters, and preconditions.
* **FR‑2  — Behavior Tree Orchestration:**  
  The orchestrator must dynamically build and execute a **Behavior Tree**, derived from the BT\_complex.mmd design, to coordinate sequential and conditional task execution.
* **FR‑3  — API‑Based Action Execution:**  
  Each atomic step (e.g., PickPlaceCabinetToCounter, TurnOnStove) shall be implemented as a leaf node that performs specific actions via the **RoboCasa API**.  
  The API provides standardized functions for querying world objects, robot motion, and environmental states.
* **FR‑4  — State Validation (“Oracle”):**  
  The system must verify the success or failure of every atomic action through ground‑truth data from RoboCasa’s **Oracle** module, enabling consistent validation independent of perception noise.
* **FR‑5  — Retry Logic:**  
  The system must detect transient, recoverable errors (e.g., failed grasp or motion timeout) and automatically re‑attempt actions up to a configurable limit.
* **FR‑6  — Fallback Logic:**  
  For non‑recoverable errors (e.g., missing eggs, broken utensil), the system must safely terminate the task and trigger appropriate fallback behavior such as NotifyUser(error\_message) or Abort.
* **FR‑7  — Adaptive Perception and RL Integration:**  
  The framework must allow integration of **Reinforcement Learning** to update object‑search and navigation policies dynamically, improving task efficiency as the robot gains experience.
* **FR‑8  — Logging and Diagnostics:**  
  The system must record environment queries, action status, and Behavior Tree traversal results for debugging, performance analysis, and RL feedback loops.

**Non-functional (how the system must perform):**

* **NFR‑1  — Robustness:**  
  The system must handle execution disturbances without halting, ensuring higher stability than linear scripting approaches.
* **NFR‑2  — Modularity and Decoupling:**  
  Task definitions (.yaml) must remain independent from the orchestration engine (Python BT Runner), supporting new task insertion without code modification.
* **NFR‑3  — Adaptability:**  
  The system should operate effectively in dynamic environments where object states or locations change across episodes.
* **NFR‑4  — Scalability:**  
  The architecture must permit easy extension with new behaviors, perception modules, or additional appliances within RoboCasa.
* **NFR‑5  — Transparency and Traceability:**  
  Every decision path in the BT must be visualizable through logs and behavior‑tree viewers for debugging and explainability.
* **NFR‑6  — Simulation Performance:**  
  The architecture must maintain a balance between control‑loop fidelity and real‑time simulation throughput, ensuring reproducible experiments.
* **NFR‑7  — Testability:**  
  The design must support unit‑ and integration‑level testing through the **Oracle‑first** validation methodology, enabling standalone verification of logic before integrating visual perception.

### **2.4 Use‑Case Definition**

The task “Boil Two Eggs” was selected as an appropriate baseline because it emulates real household routines requiring multi‑stage reasoning, physical interaction with several objects, and situational decision‑making.  
It involves the following challenges:

- **Object search and recognition**, including occluded items (e.g., eggs inside the fridge);  
- **Navigation and manipulation** across multiple kitchen zones;  
- **Dynamic state management**, requiring the system to query the environment and adapt;  
- **Error handling**, such as missing ingredients or failed grasping actions;  
- **Coordination of atomic actions** through the RoboCasa API for performing realistic interactions with the environment.

### **2.5  Research Context and Integration**

This work represents the **core orchestration layer** in a larger, modular robotic framework for dynamic domestic environments.  
While other research components focus on perception pipelines, tactile feedback, or natural‑language command parsing, the **Task Execution Module** developed in this thesis provides the glue that coordinates planning, perception, and actuation.

The system leverages the **RoboCasa API** to access high‑fidelity simulation data, ensuring deterministic experimentation and reproducible evaluation of both **Behavior Tree control** and **Reinforcement Learning‑based adaptation**.  
Together, these components form a scalable foundation for future study of **generalist home robots** capable of resilient task execution in uncertain real‑world conditions.

# 3 Studiu de piață / Abordări existente

[Dezvoltare de produs] Ce soluții similare există pe piață? Care sunt limitările lor / pentru ce cazuri de utilizare sau pentru ce tip de clienți produsele existente pe piață nu răspund cerințelor? Care sunt indicatorii pe baza cărora sunt evaluate aceste produse, de către potențiali clienți, și unde sunt lipsurile/ care este oportunitatea generată de lipsurile acestea?

[Cercetare] Metode existente (sau „State of the Art“) se referă, de regulă, la nivelul curent de dezvoltare: care este starea curentă a domeniului, unde ne găsim, care este contextul. Care sunt soluțiile actuale prezente în literatura de specialitate și care sunt limitările lor? Ce direcții de explorare sunt recomandate în literatura de specialitate? Literatura de specialitate se referă la articole științifice recente, publicate în reviste cu factor de impact mare, sau în volumele unor conferințe de top, sau în cărți.

[Ambele] În încheierea acestui capitol se dorește descrierea tehnologiilor folosite în lucrare, cu alternative și cu argumente convingătoare calitative și cantitative.

Criterii pentru calificativul *Nesatisfăcător*:

* [Dezvoltare de produs] Sunt analizate superficial câteva produse de pe piață;
* [Cercetare] analiza literaturii limitată la grupuri de cercetare din România;
* [Ambele] Sunt descrise tehnologiile folosite în lucrare.

Criterii pentru calificativul *Satisfăcător*:

* [Dezvoltare de produs] Există un interviu, un client, analiza cerințelor este elaborată pe baza interviului.
* [Cercetare] analiza literaturii de specialitate din lume, fără poziționarea precisă a lucrării în peisajului domeniului studiat;
* [Ambele] Sunt descrise câteva tehnologii alternative pentru fiecare din tehnologiile folosite în lucrare. Există o argumentare referitoare la alegere.

Criterii pentru calificativul *Bine*:

* [Dezvoltare de produs] Proces iterativ pe baza unor interviuri cu mai mulți clienți, dezvoltare MVP, reevaluare cerințe;
* [Cercetare] analiza literaturii de specialitate din lume, cu poziționarea precisă a lucrării în peisajul actual al domeniului studiat;
* [Ambele] Sunt descrise tehnologii alternative. Sunt analizate cantitativ și calitativ, folosite benchmarkuri și teste efectuate de student. Analiza este rezumată prin tabele și grafice.

First, this chapter analyzes the current "product" landscape of Smart Home Automation (SHA) and Ambient Assisted Living (AAL) to identify a critical, unaddressed market gap. Second, it performs a comprehensive "research" review of the global State of the Art (SoA) in autonomous agent control paradigms, comparing and contrasting classical and modern approaches. Finally, it synthesizes these two analyses to provide a robust, quantitative, and qualitative justification for the specific technology stack—Behavior Trees, the RoboCasa simulator, and an "Oracle-first" validation strategy—as the optimal solution to address the identified problem.

## 3.1 Market Analysis and Opportunity Definition: From Brittle Automation to Robust Orchestration

This section analyzes the existing smart home market to define a clear product opportunity, following an iterative process of requirements definition and re-evaluation.

### The Current Market: Simple, Trigger-Based Automation

The current smart home and AAL market, targeting the "general population" as well as vulnerable groups, is dominated by simple, trigger-based automation. Platforms such as IFTTT ("If This, Then That"), Amazon Alexa Routines, and Google Home define the user experience. This entire paradigm is built on "simple, linear scripts". These systems excel at stateless, single-trigger, single-outcome tasks, such as "If a motion sensor is triggered, then turn on a light" or "At 7:00 AM, then start the coffee maker."

### The "Brittleness" Gap: A Market Limitation

The fundamental limitation of this market-leading paradigm is its "brittleness". This "brittleness" describes the complete failure of these systems when faced with tasks that are complex, multi-stage, stateful, and require physical interaction. The high-value tasks identified in the project's problem statement, such as "cooking" or "household cleaning," are impossible to implement with these tools.

The reasons for this failure are threefold:

1. **Lack of Statefulness:** An IFTTT script is "fire-and-forget." It cannot verify if a task succeeded. It can *trigger* a smart oven, but it does not *know* if the oven is on, if the food is present, or if the action failed.
2. **No Error Handling:** If any step in a sequence fails, the entire process breaks. There is no native logic for "Retry" or "Fallback". If a robotic arm attempts a "failed grasp" , a linear script has no mechanism to try again; it either halts or, worse, continues as if the grasp succeeded.
3. **Inability to Sequence:** These tools are poor at orchestrating long, dependent sequences. A task like cooking involves dozens of ordered steps, each with preconditions.

The market opportunity, therefore, is not for *more* automation, but for *robust* automation. The gap is the complete absence of a commercially viable, consumer-friendly execution engine capable of robustly managing failure and complexity in long-horizon physical tasks.

### Deriving Requirements: An Iterative Process for a Domestic MVP

This thesis addresses this gap by proposing an architecture for such an engine. The requirements were developed through an iterative process, as mandated by best practices for product development.

* **Iteration 1: Initial Client Need (Analysis):** The "client" is the "general population" , which seeks to reduce time spent on "trivial tasks" like "household and family care". The highest-value targets for automation are complex activities like "cooking".
* **Iteration 2: MVP Definition (Development):** To address this need, a representative task was defined to serve as the Minimum Viable Product (MVP) specification: "Boil Two Eggs". This task is ideal because it is simple to describe but complex to execute, involving navigation (fridge to counter to stove), manipulation (grasping eggs, moving a pot), state change (filling with water, boiling), and timing. The first-pass solution was to define this task as a simple, linear "procedural plan" (conceptualized as boil\_two\_eggs.yaml).
* **Iteration 3: Re-evaluation of Requirements (The Pivot):** A critical re-evaluation of this linear plan revealed it would inherit the same "brittleness"  as IFTTT. The plan might state PickPlaceCabinetToCounter(egg\_1), but it has no way to handle an "unexpected error (e.g., a failed grasp)". This realization forced a re-evaluation of the core requirements. The system must not just *follow* a plan; it must *robustly manage* the execution of that plan.
* **Opportunity Generated:** This iterative cycle generated the project's final, core requirement: to design a "robust software architecture"  that can translate a simple, high-level plan (the "what") into a fault-tolerant execution (the "how"). This execution engine must explicitly "manage... success, failure (Fallback), and retry (Retry) logic". The "product" of this thesis is the design and validation of this robust execution architecture, which directly addresses the "brittleness" gap in the current market.

## 3.2 State of the Art (SoA) in Autonomous Task Orchestration

To build this robust engine, this section analyzes the global research literature on autonomous agent control, or "task orchestration." This review positions the thesis by comparing the dominant paradigms and identifying the one best suited to solve the problem. The core research question is: "How does an autonomous agent (e.g., a robot) execute a complex, multi-step plan in a dynamic and uncertain environment?"

### 3.2.1 Baseline: Linear Scripts and Finite State Machines (FSMs)

**Linear Scripts:** This is the most basic approach, identical to the "brittle" IFTTT model. The agent executes a hard-coded sequence of commands. This approach is not reactive and has no error handling. Any "transient failure"  (e.g., a failed grasp) causes a catastrophic failure of the entire task. This paradigm is academically trivial and unsuitable for any non-deterministic environment.

**Finite State Machines (FSMs):** A classical and powerful solution, FSMs are the foundation of control logic in many systems. An FSM is a graph of states and transitions. For the "Boil Two Eggs" task , an FSM might have states like Approaching\_Fridge, Grabbing\_Egg\_1, and Moving\_To\_Counter.

* **Limitations:** While FSMs are reactive, they suffer from a critical flaw identified in world literature: **"state explosion."** A simple, 25-step "happy path" 1 for "Boil Two Eggs" 1 would require 25 states. To make this robust, *every single state* must have transitions for all possible failures. For example, the Grabbing\_Egg\_1 state would need:
  + A transition to Moving\_To\_Counter on *success*.
  + A "Retry"  transition back to *itself* on a "transient failure".
  + A "Fallback"  transition to an Abort\_Task state on a *critical failure* (e.g., egg is missing).

This must be repeated for all 25 states, creating a "spaghetti" of hundreds of transitions that are impossible to design, debug, or maintain. Furthermore, FSMs are not modular; the logic for Grabbing\_Egg\_1 is tightly coupled to its specific state, making it difficult to reuse in another task. FSMs are not scalable for complex, robust tasks.

### 2.3.2.2 Deliberative Systems: AI Planners (e.g., PDDL, HTNs)

At the opposite end of the spectrum are deliberative systems, such as AI planners using PDDL (Planning Domain Definition Language) or HTNs (Hierarchical Task Networks). These systems do not follow a pre-defined script; they *generate* a plan from scratch, given a description of the world, available actions, and a final goal (e.g., "Goal: boiled\_eggs\_in\_bowl").

* **Limitations:** This "open-loop" approach has significant drawbacks for real-time execution:
  1. **Computational Cost:** Generating an optimal plan for a complex task is computationally expensive and can be slow, making the agent slow to react.
  2. **Lack of Reactivity:** The agent generates a plan and *then* tries to execute it. If an unexpected event occurs (e.g., a "failed grasp" ), the world state no longer matches the plan. The entire plan is invalidated, and the agent must *stop and re-plan from scratch*, which is highly inefficient.
  3. **"Black Box" Problem:** The generated plans, while optimal, can be non-intuitive to a human operator, making them difficult to debug or trust.

### 3.2.3 The Hybrid Solution: Behavior Trees (BTs)

Behavior Trees (BTs) have emerged in the last decade, primarily from the video game industry (e.g., in *Halo*), as the state-of-the-art *hybrid* solution that balances the reactivity of FSMs with the modularity of planners. They are now widely adopted in robotics, including in ROS (Robot Operating System).

A BT is a directed acyclic graph, ticked at a high frequency (e.g., 30Hz), where parent nodes control the execution of their children. The key innovations of BTs are:

1. **Modularity and Reusability:** The "leaves" of the tree are actions (e.g., PickPlaceCabinetToCounter)  and conditions (e.g., Check eggs >= 2?). These are implemented as independent, black-box modules. This means the PickPlace leaf can be reused in any number of tasks without modification, which is a major advantage over FSMs.
2. **Native Failure Handling:** This is the most critical advantage. In a BT, failure is not an *exception* to be handled; it is a *control flow mechanism*. Actions can return , , or .
   * **Fallback Logic:** A "Selector" or "Fallback" node  executes its children in order until one returns . This is the *native* implementation of failure handling.
   * **Retry Logic:** A "Retry"  decorator node simply re-ticks its child if it returns , automatically implementing the required retry logic.
3. **Readability and Debuggability:** The hierarchical, visual nature of a BT  is far easier to understand, debug, and modify than the "spaghetti" of a complex FSM.
4. **Reactivity:** Because the tree is "ticked" rapidly from the root, it is constantly re-evaluating the world state, making it highly reactive to real-time events.

BTs directly solve the "brittleness"  problem by design, providing a scalable, modular, and robust framework for task orchestration.

**3.2.3.1 Behavior Trees with Reinforcement Learning (BT‑RL Hybrid Models)**

Recent research trends (e.g., 2023–2025) explore hybrid architectures that combine the structured decision logic of Behavior Trees with the adaptive optimization capabilities of Reinforcement Learning. In such systems, the high‑level BT governs task sequencing and safety, while RL modules fine‑tune low‑level control parameters such as grasp force, trajectory choice, or exploration policies. This integration preserves BT transparency and reactivity while increasing adaptability to dynamic environments. The architecture proposed in this thesis aligns with these trends, preparing the BT framework to incorporate RL modules for perception or motion adaptation in future development stages.

### 3.2.4 Precise Positioning of This Thesis

This thesis is *not* proposing a new, fundamental control paradigm. Instead, it positions itself at the critical, modern intersection of **Behavior Tree orchestration** and **high-fidelity domestic simulation**.

While the literature has firmly established BTs in robotics (primarily for navigation or simple pick-and-place), this work is distinct in its application. It focuses on a **long-horizon, complex, multi-stage domestic task** ("Boil Two Eggs") , which involves a much greater variety of actions, locations, and state changes than typical BT benchmarks.

The primary contribution is the **design and validation of a specific BT-based architecture** that translates a high-level, human-readable plan (the conceptual boil\_two\_eggs.yaml ) into a robust execution. This architecture explicitly demonstrates the management of "Fallback" and "Retry" logic  for this class of domestic tasks. This work bridges the gap between the "brittle" market solutions (IFTTT)  and the overly complex, non-reactive academic planners (PDDL) by demonstrating a practical, robust, and testable framework for the next generation of smart home robotics.

## 3.3 Justification of Employed Technologies and Analysis of Alternatives

This section provides a detailed qualitative and quantitative justification for the three key technological pillars of this thesis, satisfying the requirement to analyze alternatives. The chosen stack is a co-dependent set of solutions:

1. **Orchestration Paradigm:** Behavior Trees
2. **Simulation Environment:** RoboCasa
3. **Validation Strategy:** "Oracle-First"

### 3.3.1 Orchestration: Behavior Trees vs. FSMs vs. Linear Scripts

The choice of Behavior Trees is the central architectural decision, justified by its objective superiority over alternatives for this project's requirements.

**Qualitative Analysis:** The following table provides a qualitative comparison of the orchestration paradigms, summarizing the State of the Art analysis.

**Table 3.3.1: Qualitative Comparison of Orchestration Paradigms**

| Criterion | Linear Script   (IFTTT) | Finite State Machine (FSM) | Behavior Tree (BT) |
| --- | --- | --- | --- |
| **Robustness (Failure Handling)** | **Very Low.** "Brittle". A single point of failure. No native error handling. | **Medium.** Requires explicit, manually-coded error transitions for every state. Non-scalable. | **High.** Native, built-in failure handling via status and Fallback/Retry nodes. |
| **Modularity / Reusability** | **Very Low.** Monolithic. | **Low.** Actions are tightly coupled to states, making them difficult to reuse or modify. | **High.** Leaf nodes (actions) are independent,  black-box modules that can be reused across  any BT. |
| **Scalability (Task Complexity)** | **Very Low.** Cannot handle sequences or state. | **Low.** Suffers from "state explosion" as complexity increases, becoming unmanageable. | **High.** New branches can be added without  affecting others. Complexity scales linearly. |
| **Reactivity (to environment)** | **Low.** "Fire-and-forget" execution. | **High.** State transitions are directly triggered by real-time events. | **High.** "Tick-based" execution (e.g., 30Hz)  constantly re-evaluates the world. |
| **Readability / Debuggability** | **Medium** (if short). | **Very Low.** Becomes "spaghetti code" that is impossible to visualize or debug. | **High.** The visual, hierarchical tree structure  is intuitive and easy to trace. |

This analysis clearly shows that for the project's requirements—robustness, scalability, and modularity—Behavior Trees are the superior paradigm.

**Quantitative Analysis (Benchmark):** A quantitative benchmark can be established not for execution *speed*, but for *architectural complexity* as a task scales. This directly addresses the "state explosion" problem.

* Let  = the number of steps in a task. For "Boil Two Eggs," .
* Let  = the number of distinct failure modes per step. For this project,  ("Transient/Retry" and "Critical/Fallback").

Now, we can model the complexity (e.g., number of states or transitions) required to make the task robust:

* **Linear Script Complexity:** . However, its robustness is . It cannot handle .
* **FSM Complexity:** To be robust, *every* step  must have transitions for *all*  failure modes. The number of transitions (the measure of complexity) scales at . As  and  grow, the complexity explodes quadratically (or worse).
* **BT Complexity:** To make a sequence of  steps robust, it is wrapped in *one* "Fallback" node and/or *one* "Retry" node. The complexity of the logic *wrapper* is . The overall complexity of the tree still scales linearly with the number of steps, , but *it maintains full robustness*.

This quantitative analysis demonstrates that Behavior Trees are fundamentally more scalable for robust task execution than FSMs.

### 3.3.2 Simulation: RoboCasa vs. Alternatives (AI2-THOR, iGibson)

The validation of this architecture requires a high-fidelity simulation environment. Testing a "failed grasp"  or interactions with a stove is not feasible, repeatable, or safe in the real world for initial development. The choice of simulator is therefore critical.

**Analysis of Alternatives:**

* **AI2-THOR:** A popular simulator, but its primary focus is on visual perception, navigation, and testing vision-based AI. It has limited support for fine-grained, stateful manipulation (e.g., grasping) and complex object-state physics (e.g., filling a pot with water, detecting if a stove is "on").
* **iGibson / PyBullet:** These are powerful, physics-first simulators. While excellent for low-level robotics and grasp-planning, they often lack the high-level semantic "affordances" of a domestic environment. One would have to build the concepts of a "refrigerator" or "stove" from scratch.

**Justification for RoboCasa:** RoboCasa was chosen because it is **explicitly designed to solve this project's exact problem**. It is a simulation environment built for "complex, multi-stage domestic tasks".

Its key advantages are:

1. **High-Level API:** RoboCasa provides the exact API "leaves" that the Behavior Tree needs to call, such as TurnOnStove or TurnOnSinkFaucet.
2. **Object State Tracking:** It is designed for stateful tasks. The simulator provides tools (like robocasa.utils.object\_utils) to check ground-truth states, such as check\_obj\_fixture\_contact. This is essential for the "Oracle-first" validation strategy.

The MVP task ("Boil Two Eggs" ) and the simulator (RoboCasa ) are a co-dependent, perfect match. The simulator provides the *exact* functionality required to test the robustness of the Behavior Tree architecture.

**Table 3.3.2: Qualitative Comparison of Domestic Robotic Simulators**

| Simulator | Primary Focus | Strengths for "Boil Two Eggs" | Limitations for "Boil Two Eggs" |
| --- | --- | --- | --- |
| **RoboCasa** | Complex, multi-stage domestic tasks. | **High.** Native API for "stove," "sink," "water."  Strong state-tracking (object\_utils). | None. It is the ideal testbed  for this task. |
| **AI2-THOR** | Visual Navigation & Perception. | Good for navigation to fridge/stove. | **Low.** Poor support for  manipulation, grasp-failure,  or state changes  (e.g., filling pot, boiling). |
| **iGibson / PyBullet** | Physics Simulation & Grasping. | **Medium.** Excellent for low-level "failed grasp" physics. | **Low.** Lacks high-level  semantic API (e.g., TurnOnStove).   Requires building the kitchen from scratch. |

### 3.3.3 Validation: "Oracle-First" vs. End-to-End Perception

The third major design decision concerns **methodology** — the strategy selected to validate the proposed Behavior Tree–based architecture.  
This thesis adopts an **“Oracle‑First” validation approach**, which prioritizes logic verification and decouples perception from execution.

#### **The Alternative – End‑to‑End Perception**

A conventional, end‑to‑end approach would integrate a full **visual perception system** (e.g., YOLO or Detectron2) at the outset.  
In such setups, the Behavior Tree’s condition nodes — for instance, Is\_Egg\_On\_Counter? — would rely directly on the outputs of the perception module.  
While appealing in theory, this coupling merges two inherently complex and error‑prone domains: reasoning and perception.

#### **The Motivation for Oracle‑First Validation**

This project deliberately avoids that approach.  
Testing both logic and perception simultaneously would compromise **testability**, a key **non‑functional requirement**.  
Two distinct challenges must be addressed separately:

1. **The Logic Problem:** Is the Behavior Tree’s orchestration logic correct? Do Retry loops and Fallback branches function as intended under controlled failure conditions?  
2. **The Perception Problem:** Can a deep‑learning vision system accurately detect relevant objects (e.g., a pot, eggs) and their spatial relationships?

When combined prematurely, any task failure becomes ambiguous — it is unclear whether it arose from flawed BT logic or a perception error.  
This makes systematic debugging and architectural validation **impossible**.

#### **The Oracle‑First Approach**

The **Oracle‑First** strategy resolves this by **decoupling** the two problems.  
Condition nodes are connected to **RoboCasa’s ground‑truth data layer (“Oracle”)**, which provides perfect, deterministic perception.  
For example, checks such as object\_utils.check\_obj\_fixture\_contact() deliver 100 % accurate environmental feedback.

This design offers several advantages:

- Enables **deterministic, repeatable tests** of the Behavior Tree’s control logic.  
- Supports validation of **Retry** and **Fallback** mechanisms under precisely controlled conditions.  
- Satisfies key **non‑functional requirements** — testability, modularity, and decoupling.  
- Creates a measurable baseline before integrating uncertain sensory inputs.

#### **Relation to Future Work and Reinforcement Learning**

Once the orchestration logic is validated, the Oracle can later be replaced or augmented with **learned perception or adaptive RL modules**.  
In that extended configuration, RL agents would interpret noisy or incomplete sensory data and adapt decision thresholds over time — all while maintaining the BT’s safe, interpretable structure.

In summary, **the Oracle‑First methodology is a deliberate, engineering‑driven decision** that isolates and verifies the thesis’s core contribution: a **robust, fault‑tolerant orchestration architecture**.  
By postponing perception integration to future work, the project ensures that each subsystem can be validated independently, providing a rigorous foundation for future extensions toward fully autonomous, perception‑driven task execution in the RoboCasa environment.

## 3.4 Conclusion

This chapter has presented a comprehensive justification for the thesis’s focus on **robust robotic orchestration** in domestic environments.  
It combined a market‑level perspective with a global research analysis to motivate the proposed technological stack and validate its relevance.

1. **Market Analysis:**  
The evaluation of the current **smart‑home automation** market reveals a clear and persistent gap.  
Existing “products” are largely brittle, supporting only simple, stateless triggers and offering minimal adaptability.  
This creates a tangible opportunity for a new class of systems that enable **robust, state‑aware, and multi‑stage task execution**, as exemplified by the project’s MVP task — “Boil Two Eggs.”

2. **State of the Art Review:**  
The analysis of international literature on control paradigms confirms that **Behavior Trees (BTs)** represent the **current state‑of‑the‑art hybrid solution** for autonomous task orchestration.  
BTs effectively resolve the “state‑explosion” and “brittleness” problems inherent in **Finite State Machines (FSMs)** and linear scripts, while providing the **modularity, reactivity, and transparency** absent in traditional AI planners.  
This thesis is precisely positioned as a **practical implementation** of BT principles applied to long‑horizon, household tasks within dynamic domestic environments.

3. **Technology Justification:**  
The selected technology stack — **Behavior Trees**, the **RoboCasa** simulator, and the **“Oracle‑First” validation strategy** — is demonstrated to be a **deliberate, interdependent, and optimal** combination of tools addressing both functional and non‑functional requirements.  
A qualitative and quantitative benchmark confirms that BTs possess **linear scalability with full robustness**, while RoboCasa offers **high‑fidelity simulation, object‑state tracking**, and a rich **API** ideal for controlled testing.  
Finally, the Oracle‑First methodology ensures **testability and decoupling**, validating the thesis’s core contribution — a **fault‑tolerant execution engine** capable of managing complex robotic orchestration independently of perception uncertainty.

In summary, this chapter established both the **scientific grounding and technological rationale** for the proposed approach.  
The following chapter, **System Architecture and Design**, details how these technologies are integrated into a unified framework that implements and evaluates the proposed robust orchestration system.

# Soluția propusă

Capitolul conține o privire de ansamblu a soluției ce rezolvă problema, prin prezentarea structurii / arhitecturii acesteia. În funcție de tipul lucrării acest capitol poate conține diagrame (clase, distribuție, workflow, entitate-relație), demonstrații de corectitudine pentru algoritmii propuși de autor, abordări teoretice (modelare matematică), structura hardware, arhitectura aplicației.

Criterii pentru calificativul *Nesatisfăcător*:

* Descriere în limbaj natural.

Criterii pentru calificativul *Satisfăcător*:

* Descriere + diagrame de baze de date, workflow, clase, algoritmi.

Criterii pentru calificativul *Bine*:

* Descriere + diagrame de baze de date, workflow, clase, algoritmi + descrierea unui proces prin care s-a realizat arhitectura/structura soluției.

This chapter presents an overview of the proposed solution and its architectural structure.  
The system is designed as a **modular, robust software framework** capable of executing — not merely recognizing — complex, multi‑stage domestic tasks.

At its core lies an **orchestration engine** that translates a high‑level conceptual task plan into a series of robust and verifiable robotic actions executed within the **RoboCasa** simulation environment.  
The overall architecture is composed of **three functional layers**, which clearly separate **task definition**, **logical orchestration**, and **physical execution and validation**.

### **4.1  Layer 1 – Task Definition Layer (“What”)**

This layer defines what needs to be accomplished, forming the conceptual blueprint of the task.  
It includes formal, human‑readable design documents that describe both the workflow and the environment context.

* **Procedural Plan (boil\_two\_eggs.yaml)** – The high‑level workflow specifying all atomic actions, their parameters, sequential order, and required preconditions.
* **Static World Model (kitchen\_map.md)** – An entity–relationship diagram describing the household layout, object identifiers, and initial world states. Logical names such as pot, fridge\_zone, or stove\_burner are mapped to simulator IDs, ensuring traceability and consistency between definition and execution.

These elements constitute the declarative layer, providing the foundation for the orchestration and execution stages.

### **4.2  Layer 2 – Orchestration Layer (“How”)**

This layer represents the **cognitive core** of the solution — the mechanism that interprets the declarative plan from Layer 1 and executes it robustly in dynamic conditions.

The Behavior Tree (BT) engine (BT\_complex.mmd) is the architectural centerpiece.  
The BT provides structured, reactive control flow and is responsible for sequencing actions, managing preconditions, and handling errors.

* **Fallback Logic:** Models non‑recoverable failures; for instance, if the Check eggs ≥ 2? node fails, execution halts and an Abort branch is triggered.
* **Retry Logic:** Addresses transient, recoverable failures such as a missed grasp or incomplete movement. The BT retries an action for a predefined number of attempts before escalating to the fallback procedure.

This hierarchical orchestration layer ensures that errors are treated as integral parts of the control logic rather than as exceptional cases, thereby enhancing robustness and fault tolerance.

### **4.3  Layer 3 – Execution and Validation Layer (“Action”)**

This layer bridges the logical orchestration with actual simulation, hosting the **Python implementation** that interfaces directly with the RoboCasa API and ground‑truth state utilities.

* **Execution (Leaf Implementations):**  
  Each atomic BT action (e.g., PickPlaceCabinetToCounter, TurnOnStove) is implemented as a Python “leaf node.”  
  These leaves control the robot by calling low‑level APIs such as env.step(...) to manipulate joints, grippers, and environment objects.
* **Validation (“Oracle‑First” Strategy):**  
  After execution, each leaf performs an objective test using RoboCasa’s “Oracle” — a direct query to the simulator’s ground‑truth data.  
  For example, calling object\_utils.check\_obj\_fixture\_contact() verifies whether a pot was successfully placed or water was turned on.  
  The result (SUCCESS or FAILED) is returned to the BT for decision logic, enabling deterministic validation of orchestration behavior.

This design isolates control‑logic validation from perception errors and fulfills the **non‑functional requirement of testability (NFR‑3)**.  
By leveraging exact state information, researchers can confirm that the orchestration logic behaves correctly before introducing uncertainty through vision or sensor noise.

### **4.4  Design and Development Process**

The three‑layer architecture evolved iteratively through the design process, aligning directly with the requirements outlined in Chapter 2.

1. **Initial Design (“What”)** – A purely procedural, sequence‑based plan (boil\_two\_eggs.yaml) was authored to describe the target task.  
2. **Iterative Evaluation (Problem Discovery)** – Early tests revealed significant brittleness. The linear plan could not handle failed grasps, missing objects, or deviations in the environment — all violating the **robustness** requirement.  
3. **Architectural Solution (“How”)** – To overcome these issues, the project transitioned to a **Behavior Tree** architecture (BT\_complex.mmd), chosen for its native support of Fallback and Retry mechanisms.  
4. **Implementation Strategy (“Action”)** – The **Oracle‑First** validation technique was introduced to decouple perception from reasoning.  
This ensures that orchestration logic can be developed, debugged, and validated independently before integrating real‑world sensors or learned perception modules.

Collectively, these steps produced a testable and modular foundation ready for integration with advanced learning‑based perception systems.

### **4.5  Adaptive Perception and Reinforcement Learning Extension**

While the initial implementation relies on Oracle‑based validation, the architecture is explicitly designed to incorporate **adaptive perception and Reinforcement Learning (RL)** for realistic operation in partially observed environments.

#### **4.5.1  Problem Scenario – Searching for Occluded Objects**

In real domestic contexts, relevant objects are often hidden or displaced — for example, a **pot inside a cabinet** or **eggs inside a refrigerator**.  
When such conditions occur, static planning is insufficient: the robot must explore, perceive, and adapt to locate the target itself.

#### **4.5.2  Behavior Tree and RL Integration**

To handle this, perception and exploration become separate **leaf nodes** inside the BT:  
- LocatePotInCabinet, OpenFridgeAndSearchEggs, or ScanCounterForPot.  
During early validation, these nodes query the simulator’s Oracle for deterministic responses.  
In future iterations, they can be **replaced by RL‑driven modules** trained to perform active search and recognition.

The BT continues to manage **task logic, retries, and safety**, while the RL agent optimizes how to perform each sub‑goal under uncertainty — for example:

* determining optimal camera orientation or exploration trajectory;
* handling partially open cabinet doors;
* searching shelves until a reward signal indicates an object is found.

#### **4.5.3  Training and Feedback Loop**

1. The BT triggers a perception/search node.  
2. If the item is not immediately detected, an RL policy executes exploratory actions, receiving **positive rewards** for partial progress (e.g., camera view improved, object found) and **penalties** for redundant or unsafe movements.  
3. Once detection is confirmed, the node returns SUCCESS, and the BT resumes execution of the main task sequence.  
4. Over time, the RL module updates its policy parameters, enabling faster or more efficient object discovery in subsequent runs.

This hybrid logic merges BTs’ interpretability and robustness with RL’s adaptability, ensuring the system can operate effectively in open‑world and dynamically changing environments.

#### **4.5.4  Advantages and Future Applicability**

- **Scalability:** The same mechanism can generalize to locating any household object.  
- **Independence:** RL nodes can be trained separately in simulation and later integrated into the validated orchestration logic.  
- **Realism:** Simulates true domestic conditions—object displacement, occlusion, and exploration—bridging simulation and physical robot deployment.

### **4.6  Final Remarks**

The proposed architecture presents a **hierarchical, fault‑tolerant orchestration framework** for complex domestic robotic tasks.  
By clearly separating **task definition**, **orchestration**, and **execution**, and by leveraging the **RoboCasa API** together with the **Oracle data layer**, the system achieves:  
- **Robustness**, through explicit Retry and Fallback mechanisms;  
- **Modularity**, through well‑defined, layered responsibilities;  
- **Testability and Scalability**, through the Oracle‑First validation method and reusable Python leaf nodes.

Beyond reliable simulation testing, the architecture unifies **reactive orchestration**, **Oracle‑based validation**, and **adaptive learning capabilities**, forming a cohesive foundation for next‑generation household robotics.  
It not only ensures robust execution within the **RoboCasa framework**, but also remains **extensible toward real‑world deployment** through the seamless integration of perception systems and **Reinforcement Learning‑based exploration** for discovering and manipulating objects in unstructured domestic environments.

## Indicații formatare formule

Formulele matematice utilizate în document vor fi centrate în pagină și numerotate. Se vor utiliza fontul Cambria Math, de dimensiune 11. Pentru a insera o nouă ecuație, utilizați Insert > Quick Parts > AutoText > Ecuație.

Toate formulele prezente în lucrare vor fi referite în text. Exemplu: *Utilizând sistemul de Insert > Bookmark*, respectiv *Reference > Cross-reference* putem cita ecuația (1) respectiv ecuația (2), citările fiind actualizate și în urma unor adăugări/ ștergeri de ecuații, cu *Select All – Update Field*. Pentru mai multe detalii despre utilizarea acestui sistem de referire și formatare puteți consulta:

<https://www.youtube.com/watch?v=9YGTH4WrY_8>.

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

# Detalii de implementare

În plus fata de capitolul precedent acesta conține elemente specifice ale rezolvării problemei care au presupus dificultăți deosebite din punct de vedere tehnic. Pot fi incluse configurații, secvențe de cod, pseudo-cod, implementări ale unor algoritmi, analize ale unor date, scripturi de testare. De asemenea, poate fi detaliat modul în care au fost utilizate tehnologiile introduse in capitolul 3.

Criterii pentru calificativul Nesatisfăcător:

* Sunt prezentate pe scurt scheme și pseudo-cod.

Criterii pentru calificativul Satisfăcător:

* Descriere sumara a implementării, prezentarea unor secvențe nerelevante de cod, scheme, etc.

Criterii pentru calificativul Bine:

* Descrierea detaliată a algoritmilor/structurilor utilizați; Prezentarea etapizată a dezvoltării, inclusiv cu dificultăți de implementare întâmpinate, soluții descoperite; (dacă este cazul) demonstrarea corectitudinii algoritmilor utilizați.

This chapter provides in-depth implementation details of the proposed robust orchestration system introduced in Chapter 4. It covers the algorithms, data structures, and iterative development process that enabled high-fidelity simulation and validation within the RoboCasa framework. Special attention is given to technical difficulties encountered, corresponding design decisions, and the integration of Reinforcement Learning (RL) for adaptive object-search tasks such as retrieving items from cabinets or refrigerators.

#### **5.1 The Behavior Tree (BT) Runner**

The core execution engine of the implementation is the **Python‑based Behavior Tree Runner**, a dynamic interpreter that governs the runtime orchestration of the “Boil Two Eggs” task.

**Primary Responsibilities:**

1. **Initialization** – Establishes a live connection with the **RoboCasa environment**, initializing all simulation parameters (gripper, arm, and sensors). Implementation draws from the demo\_teleop.py script and custom wrappers around env.reset() and env.step().

2. **Parsing** – Reads the conceptual task plan (boil\_two\_eggs.yaml) and converts it into a hierarchical action queue. Each step is mapped to a pre‑defined leaf function.

3. **Orchestration** – Implements the logic encoded in BT\_complex.mmd. Instead of a linear for loop, the runner evaluates the BT tree node‑by‑node, waiting for one of three statuses from each leaf: **SUCCESS**, **FAILED**, or **RUNNING**.

4. **Logic Flow** – Depending on the status, the runner decides execution flow:  
- On **FAILED**, it checks the node type and triggers either a **RETRY** decorator (transient failure) or a **FALLBACK** branch (critical failure).  
- On **SUCCESS**, it resumes normal sequence.  
- **RUNNING** status is managed via tick‑based polling (30 Hz) to maintain reactivity.

This modular engine encapsulates the full decision‑making logic and forms the backbone of the orchestration layer.

#### **5.2 Key Data Structures: The "Leaf" Implementation**

The **leaf nodes** constitute the physical execution layer linking decision logic to simulation reality. Each node is implemented in Python as a function following a standard interface:

class LeafNode:

def \_\_init\_\_(self, name):

self.name = name

def tick(self):  # Execute the action

result = self.perform\_action()  # Validate success using Oracle data

return self.validate(result)

def validate(self, result):

if result:

return "SUCCESS"

else:

return "FAILED"

#### **5.2.1  Bridging the RoboCasa API**

One of the most challenging technical tasks was mapping between high‑level symbolic actions and the low‑level RoboCasa API, which exposes physics‑based primitives rather than abstract commands.  
For instance:

- The action PickPlaceCabinetToCounter(egg\_1) calls env.step() with a sequence of motion vectors, computed from RoboCasa’s coordinate transforms and kinematic solvers.  
- Validation is carried out using the **Oracle utilities** within robocasa.utils.object\_utils:  
- object\_utils.check\_obj\_fixture\_contact() verifies that the egg is placed on the counter.  
- object\_utils.check\_obj\_inside\_container() is leveraged to test fridge/cabinet placement success.

This discovery came after examining internal RoboCasa tasks such as PnPCabToCounter, which exposed usable validation functions.

### **5.3  Adaptive Reinforcement‑Learning Search Node**

A major enhancement of this system is its capacity for **adaptive exploration** via **Reinforcement Learning (RL)**, enabling the robot to find and retrieve objects even when they are not visible or are placed inside closed containers.

#### **5.3.1  Problem Definition**

In many realistic tasks, essential objects (like a pot or eggs) can be:  
- hidden behind occlusions,  
- contained inside cabinets or refrigerators, or  
- moved from their known coordinates by other agents.

Traditional scripted control fails under these conditions.  
Therefore, perception and search must be **learned explorative behaviors**, governed by RL and embedded within the existing BT architecture.

#### **5.3.2  BT–RL Integration**

Within the BT, specific perception/search nodes—LocatePotInCabinet, OpenFridgeAndSearchEggs, ScanCounterForPot—are implemented as **specialized leaves** that can operate in two distinct modes:

1. **Deterministic Oracle Mode** (for testing): queries RoboCasa’s Oracle to locate the object instantly, ensuring the orchestration logic works correctly.  
2. **RL‑Driven Exploration Mode** (for autonomous operation): invokes a trained policy model that directs the robot to locate the target by scanning, opening, and reasoning about container states.

This modular design allows seamless substitution between simulated ground‑truth lookup and learned exploration.

#### **5.3.3  RL Training Workflow**

The RL component is based on a **Q‑Learning / Double‑DQN** policy trained on environmental observations from RoboCasa.  
Each episode consists of:

1. **State Encoding:** combines proprioceptive data (end‑effector pose, joint angles) with camera‑based observations or object‑contact detectors.  
2. **Action Space:** discrete exploration primitives such as look\_left, look\_right, open\_cabinet, move\_forward\_small.  
3. **Reward Design:** positive (+10) for successful object detection/identification; small penalty (–1) for wasted motion; large penalty (–10) for collision or failure timeout.  
4. **Policy Update:** the Double‑DQN optimizer reduces overestimation bias, stabilizing learning during multi‑goal episodes.

After training, the policy model is exported (rl\_locate\_object.pt) and loaded by the corresponding leaf node during execution. Results are compared against Oracle‑based detections to quantify accuracy and robustness.

#### **5.3.4  Integration Algorithm (Pseudo‑Code)**

def LocateObjectRL(object\_name):

state = get\_env\_state()

for episode in range(MAX\_ATTEMPTS):

action = RL\_policy.select\_action(state)

next\_state, reward= env.step(action)

if object\_detected(object\_name):

return "SUCCESS"

state = next\_state

return "FAILED"

This node runs as part of the BT’s “Exploration Sequence,” which executes before the main task actions.  
If LocateObjectRL() returns SUCCESS, subsequent actions like PickPlaceCabinetToCounter proceed with updated object coordinates.

5.3.5  Observed Performance

?????

### **5.4  Reference RL Methods and Comparative Analysis**

Reinforcement Learning (RL) has become a cornerstone for **autonomous navigation and manipulation** in robotics.  
Unlike static planners or rule‑based controllers, RL agents learn adaptive policies by trial and feedback, enabling them to handle uncertainty in perception and motion — for instance, **finding objects hidden behind cabinet doors or partially out of view on a counter**, as encountered in the Boil Two Eggs task.

This section reviews several RL approaches that have demonstrated effectiveness in **pathfinding, exploration, and object‑interaction tasks** relevant to the **RoboCasa** environment.

### **Reinforcement Learning Methods Side-by-Side**

Academic research has seen various RL methods applied to smart homes.

|  |  |  |  |
| --- | --- | --- | --- |
| RL Method / Approach | Typical Use in Robotics | Strengths | Limitations/Considerations |
| **Q-Learning (Discrete)** | Used in grid‑world or discrete cell  navigation to learn simple goal‑directed policies. | Simple and  interpretable; suitable for 2D pathfinding benchmarks. | Does not scale to continuous  motion or high‑dimensional  action spaces. |
| **Deep  Q‑Network  (DQN)** | Learn navigation  and grasp policies  from visual input (e.g., pixel‑based or  state vectors). | Can learn  directly from  simulated camera  frames; handles non‑linear  state-action  mappings. | Sensitive to reward sparsity;  requires careful tuning and  stabilization techniques. |
| **Double DQN (DDQN)** | Applied to object  search and pickup  in domestic simulators (e.g., AI2‑THOR, RoboCasa). | Reduces  Q‑value over‑estimation;  stable convergence for  multi‑room  environments. | Training episodes need large  sample sets and a reliable reward design. |
| **Proximal  Policy  Optimization  (PPO)** | Widely used for robot navigation  and joint motor  control in  continuous spaces. | Efficient for  continuous  motion control; robust policy updates. | Higher computation cost;  may require fine‑tuned  hyperparameters for  RoboCasa’s physics engine. |
| **Soft Actor‑Critic (SAC)** | Used for adaptive manipulation, e.g., grasp planning and motion under  object uncertainty. | Promotes  exploration  via entropy  maximization;  works  well for  multi‑modal  tasks (like  opening a  cabinet then  grasping an  object). | Complex to tune; may  introduce slower early  learning. |
| **Hierarchical RL (HRL)** | Decomposes multi‑stage tasks like  “open cabinet →  locate pot →  grab pot → place  on stove.” | Models  temporal  abstraction  and sub‑goal  decomposition naturally for  the “Boil Two Eggs”  workflow. | Requires manual definition  of high‑level sub‑goals or options. |

### 

#### **Application to This Project**

For the Boil Two Eggs MVP task, the RL module is responsible for **navigation and visual search** behaviors that extend deterministic BT planning.  
Specifically, it assists the robot in:

1. **Locating Objects in Open View:** when an egg lies on the counter, the agent uses a vision‑based DQN policy to orient the camera and move toward the detected item center.  
2. **Exploring Occluded Zones:** if the expected object is absent, the BT activates an RL node that executes a search policy with actions like open\_cabinet, look\_shelf, move\_back, tilt\_camera\_down.  
3. **Pathfinding and Recovery:** in case of failed grasp or blocked trajectory, the RL policy re‑plans locally using PPO to adjust path waypoints until the arm regains a feasible pose.  
4. **Adaptive Learning:** during training, the agent receives dense rewards for progress (approaching target, opening fridge) and penalties for collisions or invalid actions.

Mathematically, the policy π is trained to maximize expected return under reward  r(st, at) defined as:

where r(s\_t, a\_t) encourages completion of sub‑goals ( object found, cabinet opened, egg grasped ).

#### **Chosen Method for the Current Implementation**

Among the methods discussed, **Double DQN (D DQN)** was selected for initial deployment because it offers a good balance between:

- **Robustness** in multi‑stage search tasks with limited sample sizes, and  
- **Computational efficiency** on CPU‑based simulation runs within RoboCasa.

The **DDQN policy** is embedded as a **search node** in the BT, responsible for adaptive navigation and object acquisition:

- When the egg or pot is not visible, the node activates its policy network to explore containers.  
- When the object is located, the policy stores the coordinates in a shared state object (env.shared\_state["object\_pose"]) for use by the Pickup action.  
- If the policy fails to find the object after N attempts, control returns to the BT, which executes its **Fallback Recovery Path** ( e.g., notify user or re‑start task sequence ).

This hybrid BT + DDQN model allows robust decision making when the robot must **search, reach, and operate** in complex domestic environments with variable object states.

### **Summary**

This implementation of RL for **localized pathfinding and search** enhances the Behavior Tree’s deterministic logic with adaptability and exploration.  
In the Boil Two Eggs task, this translates into a robot that can autonomously **locate hidden or displaced objects, navigate efficiently to the correct coordinates, recover from failures,** and resume the main workflow with minimal human intervention.

Together, **Behavior Tree control** and **RL‑driven search** ensure fault‑tolerant, context‑aware orchestration of long‑horizon domestic robotics tasks inside **RoboCasa**.

Tabel 1 – Sumarizare criterii

|  |  |  |
| --- | --- | --- |
| Calificativ | Criteriu | Observații |
| Nesatisfăcător | Sunt prezentate pe scurt scheme și pseudo-cod |  |
| Satisfăcător | Descriere sumara a implementării, prezentarea unor secvențe nerelevante de cod, scheme, etc. |  |
| Bine | Descrierea detaliată a algoritmilor/structurilor utilizați; Prezentarea etapizată a dezvoltării, inclusiv cu dificultăți de implementare întâmpinate, soluții descoperite; (dacă este cazul) demonstrarea corectitudinii algoritmilor utilizați. | Pot fi incluse configurații, secvente de cod, pseudo-cod, implementări ale unor algoritmi, analize ale unor date, scripturi de testare. |

**5.3 Implementation Details: Algorithms, Structures, and Challenges**

The implementation phase translates the architectural design defined in Chapter 4 into functional Python code.  
This section details the key algorithms, data structures, and engineering decisions involved in constructing the robotic orchestration engine within the **RoboCasa** simulation.

**5.3.1 Core Algorithm: The Behavior Tree (BT) Runner**

The **Behavior Tree Runner** is the central orchestration algorithm.  
It serves as a Python‑based control engine that executes, monitors, and manages all task sequences for the project’s MVP scenario, Boil Two Eggs.  
Unlike predictive machine‑learning models (such as CNN‑LSTM or DDQN) described in other domains, this engine is purely **executive** — its purpose is to run actions deterministically and handle failures gracefully.

**Algorithmic Workflow**

1. **Initialization**  
Initializes a live simulation environment using the **RoboCasa API**, via robocasa.make(...).  
The implementation is inspired by demo\_teleop.py, extended with custom wrappers for env.reset() and env.step().

2. **Parsing**  
Parses the conceptual task plan (boil\_two\_eggs.yaml) into a structured task list, defining actions, parameters, and dependencies.

3. **Execution**  
Converts the parsed plan into active **Behavior Tree** nodes (BT\_complex.mmd) and executes them sequentially.  
Each node returns a status: **SUCCESS**, **FAILED**, or **RUNNING**.

4. **Status Handling**  
The runner continuously polls tree nodes at 30 Hz (“ticks”) and receives status updates from leaf actions.

5. **Logic Branching**  
When a node returns FAILED, the runner does not crash.  
Instead, it examines the BT logic to decide whether to trigger a **Retry** branch (transient error) or a **Fallback/Abort** branch (critical error).

This BT‑based logic was chosen over linear scripts (brittle) or finite state machines (FSMs with state explosion) because it directly satisfies functional requirements FR5 and FR6: robustness, modularity, and native error handling.

**5.3.2 Key Structures: Leaf Functions and "Oracle" Validation**

The **leaf nodes** are the primary data structures that bridge logical orchestration (Layer 2) and physical execution (Layer 3).  
Each leaf is a Python function that performs a specific atomic action and verifies its result using the simulator’s ground‑truth “Oracle.”

A major implementation challenge was that the RoboCasa API lacks abstract commands such as pick\_up(pot). Control must be issued through low‑level kinematic actions.  
By inspecting the simulator’s internal source code (e.g., PnPCabToCounter), the developer discovered that task validation is performed using the utility module robocasa.utils.object\_utils. This became the foundation of the **Oracle‑based validation approach**.

Each leaf function follows a consistent two‑phase pattern:

|  |  |  |
| --- | --- | --- |
| Phase | Description | Example |
| **1. Action** | Sends low‑level commands to the  RoboCasa API (e.g., motion or gripper control). | python<br>print(f"Attempting to grasp {obj\_name}...")  <br>env.step(move\_to\_object\_action)  <br>env.step(close\_gripper\_action) |
|  | Checks task success using  RoboCasa’s ground‑truth utilities. | python<br>is\_on\_counter = OU.check\_obj\_fixture\_contact(env, obj\_name, counter)  <br>is\_gripper\_far = OU.gripper\_obj\_far(env)<br>if is\_on\_counter and is\_gripper\_far:<br> return "SUCCESS"<br>else:<br> return "FAILED" |

This action/validation scheme ensures that every physical interaction returns a deterministic status. As a result, the BT receives reliable feedback, fully enabling the **Retry** and **Fallback** mechanisms that define its robust behavior.

### **5.3.3  Implementation Challenges and Staged Development**

The project followed a staged development workflow to progressively uncover limitations and validate solutions.

#### **Challenge 1 – API Discovery**

- **Difficulty:** At the start, the RoboCasa API was not documented for external task control, making it unclear how to command the robot or verify outcomes.  
- **Solution:** Through source‑code analysis of the environment’s internal tasks (e.g., PnPCabToCounter), hidden functions like \_check\_success() were identified. This led to the discovery of the robocasa.utils.object\_utils module, which became the foundation for the **Oracle‑First validation strategy (FR4)**.

#### **Challenge 2 – Testability (“Oracle‑First” Strategy)**

- **Difficulty:** Testing a fully autonomous agent requires both logical reasoning (what to do) and perception (what is seen). Errors in perception can be mistaken for logic failures, making debugging impractical.  
- **Solution:** The system adopted a strict **Oracle‑First** development methodology. By feeding the BT with perfect ground‑truth sensor information from the simulator, the orchestration logic could be implemented and validated independently of real computer vision. This ensures that retry and fallback mechanisms are verified precisely before introducing noise from visual perception.

#### **Challenge 3 – Testing Robustness**

- **Difficulty:** Demonstrating that the Retry and Fallback logic works under failure conditions — the defining feature of the BT architecture.  
- **Solution:** Dedicated test scripts were created:  
   - test\_fallback.py modifies kitchen\_map.md to remove one egg, triggering the Check eggs ≥ 2? node and verifying the Fallback branch.  
   - test\_retry.py mocks leaf\_PickPlaceCabinetToCounter() to return FAILED once, confirming the Retry decorator re‑invokes the action and then continues successfully.  
   - Additional validation cases simulate transient grasp errors and blocked paths to test adaptive response before activating the Reinforcement‑Learning (DDQN) search module introduced in Section 5.4.

### **5.3.4  Relation to Adaptive Exploration (RL Extension)**

Although the Behavior Tree core executes deterministic plans, it is architected to handle adaptive learning modules that allow the robot to find and manipulate objects even in uncertain conditions (e.g., inside cabinets or behind fridge doors).  
Integration with the **RL Search Nodes** follows the same Action/Validation protocol: a BT node triggers a Double DQN‑based policy for localized pathfinding. If the learned policy finds the target and returns coordinates, the Oracle confirms valid visual detection and the BT proceeds to the execution phase.

This modular design ensures the same logic hierarchy remains valid whether validation is derived from perfect simulator data or learned perception models.

### **5.3.5  Summary**

The implementation phase translates the system’s abstract architecture into a robust, working prototype within RoboCasa.  
Key deliverables include:   
- the **Behavior Tree Runner**, providing real‑time orchestration and failure handling;  
- the **Action/Validation leaf pattern**, ensuring deterministic feedback; and  
- an iterative testing framework that verifies Retry and Fallback behavior under  controlled errors.

Together, these components provide a verified platform on which adaptive RL modules can later be layered, allowing the robot to autonomously **search, navigate, and recover** from unexpected states such as **hidden objects, moving targets, or partial occlusion** in realistic domestic environments.

# Bibliografie

* Trebuie respectat **un singur standard** de trimiteri bibliografice (citare), **dintre** următoarele alternative:
  + APA (<http://pitt.libguides.com/c.php?g=12108&p=64730>)
  + IEEE (<https://ieee-dataport.org/sites/default/files/analysis/27/IEEE%20Citation%20Guidelines.pdf>)
  + Harvard (<https://libweb.anglia.ac.uk/referencing/harvard.htm>)
  + Cu numerotarea referințelor în ordine alfabetică sau în ordinea apariției în text (de exemplu, stilul cu numere folosit de unele publicații ACM - <https://www.acm.org/publications/authors/reference-formatting>)
* Toate referințele din acest capitol trebuie să fie referite în text. Exemple:
  + [Articol jurnal]: [2];
  + [Articol conferință]: [3];
  + [Weblink]: [5]
  + [Application report] [6]

|  |  |
| --- | --- |
| [1] | J. Silva-Martinez, "ELEN-325. Introduction to Electronic Circuits: A Design Approach," 2008. [Online]. Available: <http://www.ece.tamu.edu/~spalermo/ecen325/Section%20III.pdf>. |
| [2] | H. Baali, H. Djelouat, A. Amira and F. Bensaali, "Empowering Technology Enabled Care Using IoT and Smart Devices: A Review," *IEEE Sensors Journal,* vol. 18, no. 5, pp. 1790-1809, 2018. |
| [3] | A. Haroon, S. Akram, M. A. Shah and A. Wahid, "E-Lithe: A Lightweight Secure DTLS for IoT," in *IEEE 86th Vehicular Technology Conference (VTC-Fall)*, Toronto, 2017. |
| [4] | A. K. Jain and R. C. Dubes., Algorithms for Clustering Data, Upper Saddle River: Prentice-Hall, Inc., 1988. |
| [5] | "Kernel panic! What are Meltdown and Spectre, the bugs affecting nearly every computer and device?," techcrunch.com, 2018. [Online]. Available: https://techcrunch.com/2018/01/03/kernel-panic-what-are-meltdown-and-spectre-the-bugs-affecting-nearly-every-computer-and-device. [Accessed 14 02 2018]. |
| [6] | E. Rogers, "Understanding Buck-Boost Power Stages in Switch Mode Power Supplies," Texas Instruments, 2007. |

* NU utilizați referințe la Wikipedia sau alte surse fără autor asumat.
* Pentru referințe la articole relevante accesibile în web (descrise prin URL) se va nota la bibliografie și data accesării.
* Mai multe detalii despre citarea referințelor din internet se pot regăsi la:
  + <http://www.writinghelp-central.com/apa-citation-internet.html>
  + <http://www.webliminal.com/search/search-web13.html>
* Note de subsol se utilizează dacă referiți un link mai puțin semnificativ o singură dată; Dacă nota este citată de mai multe ori, atunci utilizați o referință bibliografică.
* Dacă o imagine este introdusă în text și nu este realizată de către autorul lucrării, trebuie citată sursa ei (ca notă de subsol sau referință - este de preferat utilizarea unei note de subsol).
* Referințele se pun direct legate de text (de exemplu „KVM [1] uses“, „as stated by Popescu and Ionescu [12]”, etc.). Nu este recomandat să folosiți formulări de tipul „[1] uses”, „as stated in [12]“, „as described in [11]“ etc.
* Afirmațiile de forma „are numerous“, „have grown exponentially“, „are among the most used“, „are an important topic“ trebuie să fie acoperite cu citări, date concrete si analize comparative.
  + Mai ales în capitolele de introducere, „state of the art“, „related work“ sau „background“ trebuie să vă argumentați afirmațiile prin citări. Fiți autocritici și gândiți-vă dacă afirmațiile au nevoie de citări, chiar și cele pe care le considerați evidente.
  + Cea mai mare parte dintre citări vor fi în capitolele de introducere „state of the art“, „related work“ sau „background“.
* Toate intrările bibliografice trebuie citate în text. Nu le adăugați pur și simplu la final.
* Nu copiați sau traduceți niciodată din surse de informație de orice tip (online, offline, cărți, etc.). Dacă totuși doriți să oferiți, prin excepție, un citat celebru - de maxim 1 frază- utilizați ghilimele și evident menționați sursa.
* Dacă reformulați idei sau creați un paragraf rezumat al unor idei folosind cuvintele voastre, precizați cu citare (referință bibliografică) sau cu notă de subsol sursa sau sursele de unde ați preluat ideile.

# Anexe

Anexele sunt opționale.

Ce poate intra în anexe:

* Exemplu de fișier de configurare sau compilare;
* Un tabel mai mare de ½ pagină;
* O figura mai mare mai mare de ½ pagină;
* O secvență de cod sursa mai mare de ½ pagină;
* Un set de capturi de ecran („screenshot”-uri);
* Un exemplu de rulare a unor comenzi plus rezultatul („output”-ul) acestora;
* În anexe intră lucruri care ocupă mai mult de o pagină ce ar întrerupe firul natural de parcurgere al textului.