Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Enhancing HTN Planning with Deep Reinforcement Learning for Method Selection

Sepideh Bahrami

Course of Study:	Computer Science
Examiner:	Dr. Ilche Georgievski
Supervisor:	Dr. Ilche Georgievski
Commenced:	December 2, 2024
Completed:	Jun 2, 2025

Abstract

Automated planning is a central area within Artificial Intelligence (AI), enabling intelligent behavior in domains such as cloud computing, autonomous systems, context-aware activity recognition, and smart environments. Hierarchical Task Network (HTN) planning, which decomposes complex tasks into simpler subtasks using predefined methods, has proven effective in such structured domains. However, its performance is often constrained by static method selection strategies that lack adaptability to varying planning contexts. To address this limitation, this thesis proposes a neuro-symbolic framework that integrates HTN planning with Deep Reinforcement Learning (DRL), combining the strengths of symbolic reasoning and data-driven learning. Among the available DRL algorithms, Deep Q-Learning (DQL) is particularly suitable due to its off-policy nature, batch-efficient learning, and robust generalization across symbolic planning states. These characteristics align well with deterministic and hierarchical planners, enabling offline learning from curated datasets without requiring interactive exploration. The proposed integration introduces a learning-based decision layer that improves adaptability while preserving the reproducibility and determinism of the underlying planner. The effectiveness of this approach is demonstrated through a comprehensive evaluation across planning efficiency, memory consumption, and plan quality. Results highlight the potential of reinforcement learning to enhance classical HTN systems and support intelligent decision-making in complex, structured environments.

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence. 3

API Application Programming Interface. 39

BNF Backus-Naur Form. 20

DNN Deep Neural Network. 21

DQL Deep Q-Learning. 3

DQN Deep Q-Network. 22

DRL Deep Reinforcement Learning. 3

DSL Domain-Specific Language. 20

HPDL Hierarchical Planning Definition Language. 15

HTN Hierarchical Task Network. 3

KDE kernel density estimate. 60

MCTS Monte Carlo Tree Search. 26

MDP Markov Decision Processes. 14

ML Machine Learning. 21

MSE Mean Squared Error. 49

PDDL Planning Domain Definition Language. 20

ReLU Rectified Linear Unit. 48

RL Reinforcement Learning. 14

TDG Task Decomposition Graph. 26

1 Introduction

This chapter introduces the context and objectives of the thesis. It begins by emphasizing the role of HTN planning within the broader field of AI planning and provides an overview of existing state-based HTN planners, focusing on their often static method selection strategies and the limitations these impose. To address such challenges, the chapter motivates the integration of learning-based techniques into HTN planning to enhance adaptability, scalability, and decision-making efficiency. It concludes by outlining the research questions and key contributions of the thesis.

1.1 The Significance of HTN Planning

Automated planning is a branch of AI that focuses on computing sequences of actions to achieve specific goals, based on a formal model of the environment [MBR+25]. Among planning paradigms, HTN planning is particularly notable for its strategic use of domain knowledge and hierarchical abstraction. Unlike classical planning, which relies on flat action and goal representations, HTN planning recursively decomposes complex tasks into simpler, more manageable subtasks using predefined methods, a strategy that closely resembles human problem-solving behavior [GNT04].

A key factor behind the success of HTN planning is its ability to model complex processes using hierarchical and structured representations. This expressive framework not only defines what needs to be achieved but also guides how to achieve it through procedural decomposition [GA15]. By embedding domain knowledge directly into the planning model, HTN planning enables more targeted search and efficient plan generation. Such a structured approach often improves scalability and performance by reducing the effective search space especially in domains where hierarchical control is critical. Notable examples include cloud application composition, autonomous driving, office activity recognition, and smart building environments [AGPA22; Geo22a; Geo25; GNLA17].

Taken together, the hierarchical structure, procedural guidance, and domain-level expressiveness make HTN planning a robust and scalable framework for solving complex tasks. Its alignment with human-like problem-solving and support for abstraction have led to its widespread adoption in diverse real-world settings. Nevertheless, while HTN planning provides a strong foundation, significant challenges remain, particularly in improving the quality and adaptability of task decomposition, which will be explored in the following sections.

1.2 Overview of Current Planners and Their Limitations

This thesis focuses exclusively on *state-based* HTN planners, where planning decisions are made based on an explicit state representation that is incrementally updated as primitive actions are executed. The discussion in this section is therefore limited to the capabilities and limitations of this class of planners.

Over the years, a variety of HTN planners have emerged, advancing the field through improvements in scalability, domain expressiveness, and task decomposition strategies [GA15]. Planners such as SHOP and SHOP2 have established strong baselines by applying a progression-based approach that processes tasks in execution order, using predefined methods and fixed rule ordering [NAI+03; NCLM99]. Building on this foundation, more recent planners enhance the method selection process by incorporating heuristic guidance and search-based decision-making. Frameworks such as H2O, SHOP-h, and Pyhop-m follow this direction, aiming to improve planning efficiency and reduce backtracking through more informed choices [CWY+18; SZC+21; WKK08].

Despite these advancements, most existing HTN planners continue to rely on fixed decomposition strategies, where method selection is guided by static rule ordering or predefined heuristics. While such approaches can be effective in structured domains, they generally lack mechanisms to adapt their decisions based on contextual information or accumulated experience. Even heuristic-enhanced planners that incorporate cost estimates or domain-independent distance metrics rely on manually designed heuristics that are fixed rather than learned [BBHB17; MGJC18]. Consequently, these systems are unable to improve their decomposition behavior based on prior planning experience, limiting their potential in scenarios where data-driven preferences could enhance decision quality.

While some systems have begun exploring more context-aware method selection through search-based decision support, such as Monte Carlo Tree Search in Pyhop-m or utility-based evaluations using Markov Decision Processes (MDP) [RCX09; SZC+21], these approaches still operate with limited learning capabilities and often incur substantial computational overhead. Other efforts, including HTN-Maker and its extension Q-SHOP [HKM10], rely on task-method value estimates that are independent of the planning state. This abstraction limits context sensitivity and results in fixed method selection that cannot adapt to varying planning conditions.

This landscape highlights a critical research gap: the absence of integrated, learning-based method selection mechanisms within HTN planning systems that can make informed, data-driven decomposition decisions during planning. While some prior approaches incorporate precomputed or heuristic estimates, they lack the ability to adapt based on accumulated experience or context. Bridging HTN reasoning with Reinforcement Learning (RL) techniques offers a promising path forward, enabling planners to generalize from prior experience and support context-sensitive decisions, even in previously unseen states.

1.3 Motivation for Intelligent Method Selection

Although HTN planners are effective at generating plans through hierarchical decomposition, most existing systems rely on fixed task traversal and static, predefined rule ordering during decomposition. When multiple methods are applicable for a given task, selection is typically based on fixed preferences, without evaluating the potential long-term consequences of each choice. As a result, planners may follow suboptimal decomposition paths, producing inefficient or unnecessarily long plans, particularly in domains with multiple valid alternatives. Since these planners lack contextual reasoning and do not leverage experience from past executions, their decisions remain uninformed. This motivates the development of intelligent method selection techniques that can assess alternatives more strategically, using a learning-based model to guide decisions and improve both planning efficiency and plan quality.

1.4 Research Questions and Objectives

The main goal of this thesis is to investigate whether the integration of a DRL model can improve the quality of method selection during task decomposition in the HTN planning process. This integration aims to move beyond rigid, search-based decision mechanisms toward more context-sensitive and informed planning behavior.

1.4.1 Research Questions

This study is guided by the following research questions:

- **RQ1**: How can a DRL model be effectively integrated into HTN planning to support intelligent method selection during task decomposition?
- **RQ2**: How can the planning process be formally represented in terms of states, actions, and rewards to construct a dataset that enables DRL to learn method selection behavior in HTN planning?
- **RQ3**: How does the performance of the learning-enhanced planner compare to a baseline in terms of planning efficiency, solution quality, and resource consumption?

1.4.2 Objectives

The following objectives are defined to address the above research questions:

1. Design and implement a modeling approach for the planning context, including the specification of state representation, action space, and reward function, to ensure compatibility with a DRL model.
2. Construct a training dataset derived from prior HTN planner executions to support the DRL model that learns method selection behavior.
3. Train and evaluate a DRL model capable of generalizing across different planning problems within a domain.
4. Design and implement a runtime integration interface between the trained DRL model and the HTN planner to enable intelligent method selection during task decomposition.
5. Evaluate the resulting hybrid planner against a baseline using benchmark domains and key performance metrics, including planning time, plan length, and memory usage.

1.5 Research Scope and Assumptions

The scope of this thesis is defined by a set of assumptions that guide the design, implementation, and evaluation of the proposed framework. This work focuses exclusively on HTN planning using a state-based formalism. Planning domains and problem specifications are written in Hierarchical Planning Definition Language (HPDL), with STRIPS-level expressiveness, including conjunctive goal descriptions and the ability to insert and delete facts from the state. A DRL model is employed as the learning component. These assumptions serve to narrow the research focus and define a well-scoped space for technical contributions. A detailed explanation of these design assumptions is provided in Chapter 4.

1.6 Research Contributions

This thesis advances the field of HTN planning by demonstrating that intelligent method selection can be achieved through the integration of a learning-based decision model into a domain-independent planner. It introduces a neuro-symbolic approach that incorporates DRL-driven decision-making into the method selection process, enabling more context-sensitive decomposition behavior. A key contribution lies in the formalization of the planning context into structured representations of states, actions, and rewards, which supports the application of DRL within the HTN framework. In addition, the thesis develops a modular integration architecture that bridges the DRL model with the HTN planner, allowing learned decision policies to be applied dynamically at runtime. Finally, the proposed approach is empirically evaluated in a classical planning benchmark domain, demonstrating measurable improvements over a baseline planner in terms of planning efficiency, memory consumption, and plan quality.

1.7 Thesis Structure

The remainder of this thesis is structured as follows:

- **Chapter 2 – Background:** Provides the theoretical foundations necessary to support the proposed approach, including key concepts in HTN planning and DRL, as well as relevant intersections between the two.
- **Chapter 3 – Related Work:** Reviews existing approaches to hierarchical planning and learning-based method selection, identifying their limitations and establishing the motivation for this research.
- **Chapter 4 – Proposed Approach:** Presents the overall design rationale, including the conceptual architecture and key components that enable the integration of learning mechanisms into the planning process.
- **Chapter 5 – Implementation:** Describes the practical realization of the proposed framework, including data preparation, model training, and the integration of learning-based decisions into the planning system.
- **Chapter 6 – Evaluation:** Reports on the empirical evaluation of the approach using a classical planning benchmark, analyzing performance across key metrics and discussing implications of the results.
- **Chapter 7 – Conclusion and Future Work:** Summarizes the contributions and findings of the thesis and outlines possible directions for further research and development.

2 Background

This chapter provides an overview of the fundamental concepts and frameworks essential for understanding this study. It begins with a discussion of AI planning and its significance in solving complex problems across various domains. The focus then shifts to HTN planning, a prominent technique that structures problems through task decomposition. Among its variants, particular attention is given to SH, a domain-independent, state-based approach. In addition, the chapter addresses the formulation of planning problems and domains, with an emphasis on their representation in HPDL, and introduces DRL along with its variant DQL, which support enhanced decision-making within the HTN planning framework.

2.1 AI Planning

Planning involves reasoning about actions, their effects, and the constraints imposed by the environment to achieve specific objectives. Rather than generating arbitrary sequences of actions, the focus is on finding efficient, goal-oriented solutions by exploring various courses of action and estimating their expected outcomes. This reasoning capability distinguishes advanced systems from simple reactive ones, allowing them to operate effectively in complex and dynamic settings. For example, in a delivery task, the system must account for factors such as time windows, resource availability, and vehicle capacities, making trade-offs to ensure timely and cost-effective deliveries. The inherent complexity lies in navigating this multidimensional space of possibilities while adhering to given constraints and optimizing outcomes [GNT04].

Building on the foundations of planning, AI planning, also referred to as *automated planning*, is a subfield of AI that leverages computational techniques to address complex problems across various domains. It focuses on devising procedural courses of action for declaratively described systems to achieve specified goals while optimizing performance. AI planning is particularly useful in scenarios where problems can be described declaratively and domain knowledge is available and valuable. It is also well suited for settings that require structured task reasoning or where interpretability and control over the planning process are important. It has been widely applied in domains such as robotics, manufacturing, logistics, transportation, and space. For instance, in logistics, it optimizes scheduling and routing to meet time and resource constraints [Soh19].

Notable applications of AI planning include requirement engineering, where it helps analysts understand the impact of stakeholder preferences on design decisions; web service composition, which enables the customization of services based on user preferences; and domains such as robotics and logistics, where it optimizes scheduling, routing, and task allocation to meet time and resource constraints. Other applications include future state projection in the energy domain, where it forecasts long-term outcomes such as oil production and pricing; and enterprise risk management, which assists financial organizations in identifying emerging risks from news and social media data. These applications often highlight challenges such as integrating domain knowledge into AI planners and addressing novel research problems. Despite these challenges, AI planning continues to offer innovative solutions, with the potential to drive advancements across AI technologies and practical applications, benefiting both immediate needs and the broader research community [Soh19].

2.2 HTN Planning

HTN is an AI planning technique designed to handle complex tasks by breaking them down into simpler subtasks. Compared to classical planning approaches, HTN planners excel in leveraging sophisticated knowledge representation and reasoning [GNT04]. Several well-known HTN planning systems, such as Nonline, SPIE-2, O-Plan, UCMP, NOAH, and SHOP2, have been developed for diverse applications. For instance, SPIE-2 has been applied in air-campaign planning, O-Plan in modeling biological pathways, SHOP2 in crisis management and logistics, Nonline in construction planning, and NOAH in production planning [GA15].

There are two types of constructs at the core of HTN planning: primitive and compound tasks [GA15]. Primitive tasks are equivalent to the actions in classical planning. They are the basic operations that can be executed directly when their preconditions are met in the current state. In contrast, compound tasks represent more complex activities that cannot be performed in a single step. Instead, they must be decomposed into a hierarchy of subtasks using predefined decomposition methods. A single compound task may be broken down by several different methods as specified in the planning domain, while there is typically a one-to-one correspondence between operators and primitive tasks, ensuring that every executable action has a well-defined effect on the environment [HBBB21].

HTN planners are commonly classified according to their search space into two categories: plan-based and state-based. In plan-based HTN planning, the focus is on refining a task network until a fully specified, executable plan is achieved. By contrast, state-based HTN planning, which is particularly relevant for our work, integrates task decomposition directly with state transitions. In this approach, planning starts from the initial state with an empty plan. As compound tasks are decomposed, primitive tasks are executed immediately, causing state transitions. This execution model results in explicit state transitions throughout the planning process. A distinguishing feature of this approach is the early-commitment strategy, in which the planner selects the first applicable method for a task based on the current state, without exploring alternatives unless backtracking is required. This strategy simplifies the management of task interactions and reduces the complexity of the search space, as described in [GA15]. The state-based approach offers a natural framework for this thesis, enabling efficient handling of evolving scenarios through direct state manipulation.

2.2.1 Formal Definitions

HTN planning has been formalized in several works, each defining the essential components and structures of the paradigm [BAH19; GA15; Geo15; GNT04]. These formalizations offer a foundational understanding of how hierarchical planning problems are represented and solved. This section provides an overview of selected definitions from the literature to familiarize the reader with the core concepts of HTN planning.

Definition 2.2.1.1 (Predicate)
*A predicate $p \in Q$ represents a relation among objects in the environment. It is defined by a **predicate symbol** and a sequence of **terms**, formally expressed as $p = \langle symbol(p), terms(p) \rangle$, where:*

- *Q is a finite set of predicates.*
- *Each term is either a constant $c \in C$ or a variable $v \in V$.*
- *C is a finite set of constants, and V is an infinite set of variables.*
- *A ground predicate is a predicate in which all terms are constants, representing a concrete fact about the world.*

Definition 2.2.1.2 (State)
*A $state = \langle p_1, p_2, \ldots, p_n\rangle$ represents the properties of the environment at a specific moment in time, described by a set of **ground predicates**, where each p_i is a ground predicate. Under the Closed World Assumption, any predicate not explicitly included in the state is assumed to be false.*

Definition 2.2.1.3 (Primitive Task)
A primitive task is represented as an expression $t_p(\tau)$, where $t_p \in T_p$ denotes a symbol from a finite set of primitive tasks T_p, and $\tau = \langle \tau_1, \ldots, \tau_k\rangle$ is a sequence of terms. Each primitive task is represented by a planning operator.

Definition 2.2.1.4 (Operator)
*An **operator** $o \in O$ represents an action that can be executed in the environment. It is defined as a triple: $o = \langle p(o), pre(o), eff(o)\rangle$, where*

- *$p(o)$ is a primitive task associated with the operator.*
- *$pre(o) \subseteq 2^{\mathcal{Q}}$ denotes the **precondition**, a set of predicates that must hold before the operator can be applied.*
- *$eff(o) \subseteq 2^{\mathcal{Q}}$ represents the **effect**, which describes how the environment changes after the operator is executed.*

The precondition is divided into two subsets:

- *$pre^+(o)$ specifies the **positive preconditions** that must hold in the current state.*
- *$pre^-(o)$ defines the **negative preconditions** that must not hold in the current state.*

*An operator o is **applicable** in a state s if $pre^+(o) \subseteq s$ and $pre^-(o) \cap s = \emptyset$. Applying the operator transforms the current state s into a new state s' as follows: $s' = s[o] = (s \setminus eff^-(o)) \cup eff^+(o)$*

where $eff^+(o)$ and $eff^-(o)$ denote the set of predicates added and deleted by the operator, respectively.

Definition 2.2.1.5 (Compound Task)
A compound task is an expression $t_c(\tau)$, where $t_c \in T_c$ represents a symbol from the finite set of compound task symbols T_c, and $\tau = \langle \tau_1, \ldots, \tau_k\rangle$ is a sequence of terms. Compound tasks require further decomposition into subtasks and cannot be executed directly.

Definition 2.2.1.6 (Task Network)
A task network tn is represented as $\langle T, \prec\rangle$, where T is a finite set of tasks and $\prec$ is a partial order on T.

Definition 2.2.1.7 (Method)
*A **method** m is a pair $\langle c(m), tn(m)\rangle$, where:*

- *$c(m)$ is a **compound task** that the method aims to decompose.*
- *$tn(m)$ is a **task network** that represents the decomposition of the compound task into a set of subtasks along with their ordering and constraints.*

Methods define how a compound task can be broken down into smaller tasks to achieve the desired goal.

Definition 2.2.1.8 (Decomposition)
Given a task network $tn = \langle T, \prec\rangle$ and a method m for a compound task $t = c(m)$, the method m decomposes the task network tn into a new task network tn' as follows:

$$tn' = ((T \setminus \{t\}) \cup T_m, \prec \cup \prec_m \cup \prec_D)$$

where $\prec_D$ is the set of dependencies defined by:

$$\prec_D := \{(t_1, t_2) \in T \times T_m \mid (t_1, t) \in \prec\} \cup \{(t_1, t_2) \in T_m \times T \mid (t, t_2) \in \prec\}$$

Definition 2.2.1.9 (Domain)
An HTN planning domain d is a pair $D = \langle Q, V, T \rangle$, where:

- *Q is a set of predicates.*
- *$v = \langle v_1, \ldots, v_n \rangle$ is a set of variables, and each $v_i \in V$ range over a finite domain D_i.*
- *T is the union of primitive and compound tasks.*

Definition 2.2.1.10 (Planning Problem)
*A **planning problem** defines a specific instance of planning within a given domain. It is represented as a tuple: $\rho = \langle s_0, tn_0, d \rangle$, where*

- *s_0 is the initial state of the environment.*
- *tn_0 is the initial task network.*
- *d is the domain, which defines the available actions and constraints within the planning problem.*

Definition 2.2.1.11 (Solution)
Let P be an HTN planning problem. A sequence of operators $o_1, o_2, \ldots, o_n$ is a solution to P if and only if there exists a task $t \in T_0$, where $tn_0 = \langle T_0, \prec_0 \rangle$, such that for all $t' \in T_0$, $(t, t') \in \prec_0$, and the following conditions hold:

- *If t is primitive and applicable in the state s_0, then the sequence $o_2, \ldots, o_n$ is a solution to P, where the task network is $tn_0 \setminus \{o_1\}$ and the state is updated to $s_0[o_1]$.*
- *If t is compound, there exists a task decomposition in s_0 such that the sequence $o_1, \ldots, o_n$ is a solution to P, with the task network evolving as $tn_0 \xrightarrow{D} tn'$.*

2.2.2 Hierarchical Planning Definition Language (HPDL)

HPDL is a formal language employed by the SH planner. It is designed to extend the syntax and semantics of Planning Domain Definition Language (PDDL) 2.1 level 3 [FL03] to support hierarchical task planning. While retaining foundational constructs of PDDL for defining objects, types, predicates, and primitive actions, HPDL introduces critical enhancements, including the representation of tasks at multiple levels of abstraction and decomposition methods to specify alternative ways to achieve compound tasks. Each task can be decomposed into subtasks using methods, which define the conditions under which they are applicable through preconditions. This enables flexible and context-dependent planning based on the current state. Unlike PDDL, which defines goals as a set of terminal state facts, HPDL formulates goals as tasks to be accomplished, shifting the focus from static states to actionable objectives [VFP20]. The syntax of HPDL, based on an extended Backus-Naur Form (BNF), is detailed in [Geo13].

2.2.3 SH Planner

The SH planner is an open source HTN system designed to support complex task decomposition and integrate advanced reasoning capabilities. It serves as the core planning system in this study and performs plan synthesis using depth-first search over an initial task network. SH operates on HPDL-encoded planning problems and domains, which are translated into structured Scala object representations to enable hierarchical problem-solving without relying on explicit code generation. Additionally, with minor modifications, SH allows planning problems to be defined directly in Scala through a Domain-Specific Language (DSL), eliminating the need for an external transformation step [GPAA24].

SH provides support for various logical constructs, making it a highly expressive planner for hierarchical planning problems. It allows typed parameters for operators and tasks, supports conjunction, disjunction, implication, and universal quantification in preconditions, and incorporates numeric fluents in both preconditions and effects. Additionally, it supports universally quantified effects and axioms, further extending its reasoning capabilities [Geo15]. The planner follows a modular architecture, consisting of multiple layers that handle problem parsing, plan generation, grounding, and external service integration. One of its key features is runtime grounding, where predicates and task instances are instantiated dynamically during the planning process instead of being precomputed, improving efficiency in large-scale problems [GPAA24].

Due to its scalability and robustness, SH has been successfully applied in various fields, including autonomous systems, smart environments, and cloud deployment. Its flexible design allows researchers and practitioners to customize and extend the system based on specific domain requirements [GPAA24]. With its integration of HPDL and structured problem-solving mechanisms, SH represents a step toward a unified and well-defined language for state-based HTN planners [Geo15].

2.3 Deep Reinforcement Learning (DRL)

RL is a subfield of Machine Learning (ML), which in turn is a branch of AI concerned with developing agents that improve their performance through experience. In the RL paradigm, an agent learns to make decisions by interacting with an environment and receiving feedback in the form of scalar rewards. At each time step, the agent observes the current *state* of the environment, selects an *action*, receives a corresponding *reward*, and transitions to a new state. This sequence of interactions forms an *episode*, and the objective of the agent is to learn a *policy*, a mapping from states to actions, that maximizes the expected cumulative reward, also referred to as the *return*. To achieve this, many RL algorithms utilize *value functions*, such as the state-value function $V(s)$ and the action-value function $Q(s, a)$, to estimate the expected return from particular states or actions. These interactions are typically modeled using a MDP, which provides the mathematical structure for learning and decision-making under uncertainty [DDD+20].

Building upon these foundations, DRL integrates RL with Deep Neural Networks (DNNs) to enable scalable learning in high-dimensional or continuous environments. Instead of representing policies or value functions in tabular form, DRL approximates them using neural networks, enabling agents to generalize across large state spaces and learn directly from raw input features such as images or structured symbolic representations. This capability has allowed DRL agents to achieve superhuman performance in tasks such as Go and Atari games and has expanded its applicability to domains such as robotics and autonomous systems [DDD+20].

In the context of this study, a particularly relevant setting is *offline* DRL, where the agent learns exclusively from a fixed dataset of past interactions rather than engaging in online exploration during training. While this approach is suitable for environments where interaction is costly, risky, or impractical, it introduces specific challenges such as distributional shift between the behavior policy in the dataset and the learned policy. To address these issues, offline DRL algorithms employ strategies such as conservative value estimation, support-aware regularization, and uncertainty-aware updates to promote robust policy learning [PMC23].

Figure 2.1 illustrates the standard RL framework, which also forms the foundation of the DRL paradigm.

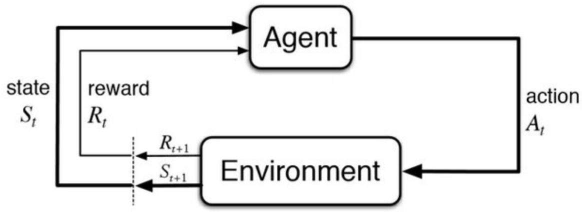


Figure 2.1: The standard RL framework: interaction between agent and environment. Adapted from [AR18].

2.4 Deep Q-Learning (DQL)

Q-learning is a widely used model-free RL algorithm, meaning it does not require a model of the environment's transition dynamics and instead learns directly from experience. It seeks to learn the optimal action-value function, $Q(s, a)$, which estimates the expected cumulative reward an agent can obtain by taking action a in state s and thereafter following the optimal policy [AR18; WD92]. Q-values are updated iteratively based on observed rewards and estimated future returns using the following update rule [AR18]:

$$Q(s, a) = Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{2.1}$$

In this equation, α is the *learning rate*, which controls the extent to which newly acquired information overrides previous estimates. The term r denotes the immediate *reward* for action a in state s. γ is the *discount factor*, which balances the importance of short-term versus long-term rewards. s' is the resulting next state, and a' denotes the set of possible actions in that state. The term $\max_{a'} Q(s', a')$ estimates the value of the best possible action in the next state. This update rule allows the agent to adjust its Q-values based on both immediate and anticipated future rewards. Under certain conditions, repeated application of this rule enables convergence to the optimal action-value function [AR18]. While this tabular approach is theoretically sound and converges under suitable conditions, it becomes impractical in high-dimensional environments due to its lack of generalization and significant memory requirements [FWXY20; MKS+15].

To overcome these challenges, DQL extends classical Q-learning by integrating it with DNNs, resulting in Deep Q-Networks (DQNs) that approximate the action-value function through neural networks. Instead of maintaining a Q-table, a DQN learns a continuous mapping from state-action inputs to expected returns. This allows the agent to generalize across high-dimensional or symbolic state spaces and scale effectively in structured decision-making domains [DDD+20; MKS+15].

Although DQL retains the core Bellman-based update rule introduced in Equation 2.1, its training process incorporates two key mechanisms to enhance stability. First, a *target network* is maintained as a periodically updated copy of the Q-network to mitigate instabilities caused by shifting targets during bootstrapping. Second, an *experience replay buffer* stores past transitions, from which uncorrelated mini-batches are randomly sampled during training. This reduces the correlation between consecutive updates and improves sample efficiency [MKS+15].

Beyond its core learning mechanism, DQL exhibits a set of defining properties, three of which are particularly relevant for this study. First, it is inherently *model-free*, meaning it does not require prior knowledge about the transition or reward functions of the environment and instead learns directly from interaction data [MKS+15; WD92]. Second, as an *off-policy* method, DQL learns the optimal policy independently of the behavior policy used to generate the training data [AR18]. This allows it to effectively leverage previously collected trajectories, even if they were generated by a different or suboptimal policy.

Third, its training procedure is naturally compatible with *batch-style updates* using mini-batches, making it well-suited for offline settings where learning must occur from fixed datasets [EGW05; MKS+15]. These properties collectively distinguish DQL as a scalable and flexible framework for learning value-based policies in both online and offline environments.

3 Related Works

There are several studies that have explored strategies to enhance the effectiveness of HTN planners. Among these, this chapter specifically outlines research on state-based HTN planners that incorporate heuristic guidance, search-based decision mechanisms, or learning-inspired techniques. These works aim to improve the task decomposition process through various forms of support. Reviewing this body of work helps contextualize recent advancements in the field and lays the groundwork for introducing the contributions of our proposed approach.

In a study proposed by [WKK08], a planner called H2O is introduced to combine heuristic search with HTN planning. It aims to improve task decomposition processes. Unlike traditional HTN planners such as SHOP2, which rely on predefined method orderings and backtracking upon failure, H2O integrates domain-independent heuristics to dynamically select the most suitable decomposition methods. However, H2O's reliance on static heuristic functions, such as Manhattan Distance, restricts its adaptability to complex domains and makes its performance highly dependent on the quality of the chosen heuristics. Furthermore, the absence of learning mechanisms based on past planning experiences prevents H2O from using historical data to support a more context-aware method selection strategy.

In a similar approach, [CWY+18] focus on enhancing HTN planning by integrating domain-independent heuristic search with state-based task decomposition. They propose a modified planner, SHOP-h, built upon the original SHOP framework. This modification introduces a goal state into each decomposition method and uses a heuristic-guided method selection strategy. Unlike its predecessor, SHOP-h operates deterministically by evaluating all applicable methods for a compound task using a heuristic function that estimates the distance between each method's goal state and the overall goal state. This distance is computed using domain-independent metrics, such as Manhattan or Euclidean distances. By selecting the method with the lowest heuristic value, the planner reduces backtracking and improves plan quality. However, similar to the H2O planner, SHOP-h's effectiveness remains highly dependent on the quality of the heuristic function, which can limit its adaptability and scalability in more complex domains.

In another study, [MGJC18], the authors propose a cost-optimal HTN planning framework that integrates branch-and-bound search with heuristic guidance inspired by the Anytime A* algorithm. The approach uses two heuristics: an admissible one for pruning non-promising task networks, and a dynamically weighted non-admissible heuristic to guide task selection. It also introduces two decomposition strategies namely, First Strategy and Highest Strategy. The First Strategy decomposes tasks in order, while the Highest Strategy prioritizes high-level tasks and performs better in hierarchical domains. Although the framework shows strong anytime behavior and improves plan quality over time, its reliance on manually tuned heuristics, static cost assumptions, and limited empirical evaluation restricts its applicability in diverse planning domains.

In the work of [HBBB18], a progression-based HTN planning system (also referred to as state-based) is introduced that integrates classical, domain-independent heuristics to guide method selection. Unlike many existing progression-based planners that either ignore heuristics or rely on modified models, this approach supports the use of arbitrary classical heuristics by explicitly incorporating state-based goals into HTN planning. The authors improve the standard progression algorithm by reducing unnecessary branching through fewer non-deterministic choice points and prove the soundness and completeness of their method. To enable heuristic guidance, they relax hierarchical constraints and translate parts of the HTN problem into a STRIPS-compatible form, making it suitable for traditional heuristic estimation. This translation relies on bottom-up and top-down reachability analyses, which allow the planner to estimate the cost of decompositions more effectively. However, relaxing the hierarchy can weaken the representation of structural constraints, potentially resulting in heuristic estimates that favor decompositions that are suboptimal or even invalid in the original HTN model. While the approach enhances efficiency and generality, its success depends on how well the relaxed model captures the essential structure of the planning domain.

Another contribution to heuristic-guided HTN planning is presented by [BBHB17], who propose an admissible heuristic based on a structure called the Task Decomposition Graph (TDG). Their approach computes cost estimates that represent the least-expensive way to refine abstract tasks into primitive ones, enabling admissible heuristic search within standard HTN frameworks. To improve heuristic accuracy during search, they also introduce a TDG recomputation mechanism that updates heuristic estimates after decompositions. Empirical evaluations show that while the method guarantees optimal plans, it may incur additional computational overhead due to the cost of recomputation, and in some domains, runtime increases were observed despite reductions in search space size. This work highlights ongoing efforts to strengthen HTN planners through heuristic design, providing important context for the shift toward learning-based guidance explored in this thesis.

As the field moves beyond traditional heuristic guidance, some approaches have begun to explore search-based decision support methods. Among these, Pyhop-m stands as an intermediate step between heuristic-driven planning and learning-based techniques. In this context, [SZC+21] propose Pyhop-m, an HTN planner that employs Monte Carlo Tree Search (MCTS) to incrementally build a planning tree during decomposition, helping the planner to systematically select among available methods. By simulating different task decompositions, Pyhop-m reduces the dependence on manual method ordering and helps generate better plans compared to heuristic-based baselines. However, this exploration strategy also increases the size of the search tree, leading to higher space complexity as the number of decomposition options grows. Their approach represents an effort to improve method selection in HTN planning by incorporating simulated search, with benefits in plan quality but an increase in search space complexity.

In the context of combining learning-based approaches with HTN planning, one example is [RCX09], which applies an integration using a MDP model to improve web service composition. This method enhances decision-making by modeling multiple decomposition strategies for a given task and evaluating them using non-functional properties such as cost, reliability, and response time. A policy is derived through iterative evaluation of state-action values, guided by a reward function based on fixed Quality-of-Service parameters. A threshold parameter, λ, is used to control the solution space and balance between optimistic and pessimistic cost estimations, with the quality of results depending on the selection of an appropriate threshold. While this approach formalizes method selection through policy iteration, it also incurs significant computational overhead from evaluating all possible decomposition paths. Overall, the work demonstrates how MDP-based models can support structured, reward-driven decision-making in HTN planning, particularly in domains where decomposition choices are evaluated using predefined utility metrics.

Another relevant contribution to learning-based improvements in HTN planning is presented by [HMK16], who propose HTN-Maker, an offline learning framework for automatically inducing HTN decomposition methods from observed plan traces. Their approach extracts hierarchical structures and applicability conditions without requiring prior hierarchical models, aiming to generate reusable planning knowledge from input data. While HTN-Maker focuses on learning new methods rather than optimizing method selection during planning, it illustrates the potential of offline learning strategies for enhancing hierarchical planning frameworks.

A related effort by [NFP22] introduces a planning and acting architecture where a DQL module is trained to select intermediate subgoals in a classical planning framework. This approach demonstrates the effectiveness of using DQL for goal selection in real-time environments, offering improved generalization and reduced planning time. Their system is evaluated in the Boulder Dash game environment, where subgoals correspond to reaching specific objects or locations. Although the underlying planner is classical and does not involve hierarchical decomposition, the work illustrates how reinforcement learning can successfully guide high-level planning decisions and highlights the applicability of DQL to planning systems beyond classical frameworks.

Beyond hierarchical and classical planning, DQL has also been applied to structured planning problems in other domains. For instance, [SZXD17] address the ship stowage planning problem by modeling it as a MDP and training a DQN offline to optimize container placement decisions. Their approach focuses on minimizing operational costs such as reshuffles and yard crane shifts during container loading. Although the setting involves direct action selection without hierarchical decomposition, the work illustrates the broader applicability of RL techniques to structured planning tasks. This broader context motivates further exploration into applying DQL to more abstract and hierarchical planning models, such as HTN planning.

Another example of using DQL in structured decision-making tasks is presented by [MCNF17], who focus on tactical lane-changing in autonomous driving. Their framework separates control into two levels: a low-level controller for safe lane maneuvers and a high-level DQL model for deciding when to accelerate, decelerate, or change lanes. They introduce Q-masking, a technique for integrating prior knowledge and constraints directly into the learning process by filtering infeasible actions during training. Although this work does not involve hierarchical task decomposition, it highlights the utility of DQL for learning high-level planning policies in structured environments. It also demonstrates the benefits of offline training setups in domains where low-level control is well-understood but high-level decision-making remains challenging.

Finally, a particularly relevant effort is presented by [HKM10], which explores the integration of RL within HTN planning by learning new decomposition methods and estimating their utility through offline episodic simulations. Their framework includes Q-MAKER, which induces HTN methods from plan traces and assigns initial Q-values, and Q-REINFORCE, which refines these values using Monte Carlo simulations. The final component, Q-SHOP, is a modified HTN planner that selects among learned methods based on their estimated returns. Their use of offline RL, where training is based on pre-collected planning traces, inspired our decision to adopt an offline training strategy for DQL. This connection marks an important conceptual link between their work and the present research, providing context for the distinct objectives and learning strategy pursued in this thesis.

4 Proposed Approach

This chapter develops the conceptual foundation for the neuro-symbolic approach adopted in this study. It begins with a high-level architectural perspective, introducing the main modules of the system, explaining the rationale behind key design decisions, and providing an overview of component interactions and execution logic. The discussion then turns to the principles that make a planning domain compatible with learning-based methods, followed by a description of how the planning context is modeled to support RL. Finally, the chapter outlines the design of a structured dataset based on the Medallion Architecture, a data engineering best practice.

4.1 System Architecture

This section introduces the high-level design of a hybrid planning framework that integrates a symbolic hierarchical planner with a learning-based decision-making component. The conceptual framework comprises three principal modules: a state-based HTN planner responsible for hierarchical task decomposition, a DRL model that learns to select appropriate methods based on the planning context, and an integration interface that facilitates communication between the two. The architectural design is guided by principles from the AI Planning Software Development Life Cycle [Geo23], with a focus on modularity, maintainability, and cross-component compatibility. The remainder of this section is structured as follows: Section 4.1.2 presents the roles and selection rationale for each component. Section 4.1.1 discusses the motivation behind decoupling the system into modular services. Section 4.1.3 outlines the execution logic and interaction flow among the selected components.

4.1.1 Architectural Rationale for Decoupling

A central architectural choice in the hybrid framework presented in this thesis is the decoupling of the HTN planner from the DRL model. In this design, they operate as independent components that communicate through a well-defined interface. This decision is grounded in established software engineering principles that emphasize modularity, scalability, and maintainability—values also promoted within the AI planning community as discussed in [GB21; Geo22b].

A key advantage of this decoupling is *language independence*. For instance, a state-based HTN planner such as SH, implemented in Scala, a language well-suited for building scalable, modular systems with strong static typing and functional constructs [OAC+04], can be paired with a learning model implemented in Python, which offers a rich ecosystem of machine learning libraries such as PyTorch and supports rapid prototyping and experimentation. This separation enables each component to leverage the strengths of its native language, allowing for robust planning logic and flexible model development. Furthermore, decoupled components can evolve independently, minimizing integration overhead and facilitating iterative improvement.

The decoupled design also improves *maintainability*. All learning-specific responsibilities, such as model architecture, input normalization, and vocabulary handling, are encapsulated within the Python service. This separation preserves the stability of the HTN planner and simplifies experimentation with alternative learning strategies. As a result, the system remains robust even as the learning component is refined.

Lastly, this architecture enhances *scalability*. Since the learning model is deployed as a standalone service, it can be executed on dedicated hardware, containerized using tools like Docker, or horizontally scaled using load balancers. This flexibility ensures responsiveness, particularly under high planning loads or concurrent problem-solving scenarios. This architectural choice lays the groundwork for the modular interaction design discussed in the next subsection.

4.1.2 Component Overview and Selection Rationale

We propose a hybrid planning system in which a symbolic HTN planner is augmented with a data-driven method selection model based on DRL. While the architectural framework remains conceptually independent of any particular HTN planner or learning model, this thesis instantiates it with components that align with the practical constraints and research objectives of the study.

One of the main components of the proposed framework is the planning module, which is realized using the *SH* planner. SH is a state-based, domain-independent HTN planner designed with an emphasis on modularity, generality, and formal expressiveness [GPAA24]. It operates over the HPDL formalism and supports deterministic planning, enabling the generation of reproducible task decomposition traces. These characteristics make it particularly well suited for integration in a hybrid planning framework, where consistent planning behavior and structured outputs are essential for learning-based decision support. In addition, the conceptual modularity of SH aligns naturally with the decoupled architectural design of the proposed system, allowing the planning and learning components to interact without mutual dependency while remaining semantically coordinated.

The learning component in the proposed framework is tasked with supporting intelligent method selection during planning by modeling context-dependent decisions. This role is conceptually realized using a Deep Q-Learning (DQL) approach, which exhibits several properties that align closely with the design requirements of the hybrid system [AR18; EGW05; MKS+15; WD92]. First, DQL follows a model-independent learning strategy, meaning it does not rely on predefined environment dynamics. This abstraction aligns well with structured planning settings, where such dynamics are often difficult to specify or intentionally decoupled from the learning process. Second, DQL supports learning from fixed planning traces, irrespective of how those traces were originally generated. This flexibility enables the learning component to operate independently of the planner's behavior during data collection, reinforcing the framework's offline training strategy and modular separation between planning and learning. Third, DQL supports structured learning from pre-defined fixed datasets, enabling the training process to be fully decoupled from the planner's execution. Together, these properties make DQL a conceptually well-matched choice for supporting method selection within a modular, hybrid planning framework.

The third main component of the proposed system is the integration interface, which facilitates communication between the planning and learning modules. Given the decoupled nature of the architecture, where the symbolic HTN planner and the learning-based method selector are developed in different environments, a well-defined, language-agnostic communication layer is essential. To fulfill this role, the system adopts a RESTful interface, which offers a lightweight and standards-compliant protocol for interaction between components.

A deliberate architectural choice underlying the learning strategy is the adoption of an *offline RL* paradigm. Rather than relying on continuous interaction with the planner during training, the learning model is trained on a fixed corpus of planning experiences collected independently. This decision supports the separation of concerns between planning and learning, allowing the planner to remain stateless and deterministic while the learning model improves over time without disrupting the planning pipeline. In addition, the offline setup enhances experimental reproducibility and aligns well with the goal of developing a modular framework, where training dynamics are decoupled from planning behavior. This design decision enables the use of precomputed planning traces as training input, facilitating the reuse of existing benchmark domains and reducing system complexity during learning.

Overall, while the chosen components are tailored to the goals of this thesis, the proposed framework generalizes to alternative HTN planners and learning models, provided they expose compatible interfaces and decision points for method selection.

4.1.3 Designed Interaction Logic

The interaction logic between the symbolic planner, the integration interface, and the DQL model forms a central aspect of the proposed system design. As illustrated in Figure 4.1, the architecture captures the flow of information across modular components in a hybrid planning framework. The starred components in the figure, namely the SH planner, a RESTful API, and a Deep Q-Learning model, represent the specific instantiation adopted in this thesis. However, the architecture itself remains deliberately general. As emphasized earlier, the framework is designed to accommodate alternative HTN planners and intelligent decision models, provided they expose compatible decision points and maintain interface consistency. This design-level flexibility allows the proposed architecture to serve as a reusable foundation for integrating learning-based method selection into symbolic planning pipelines.

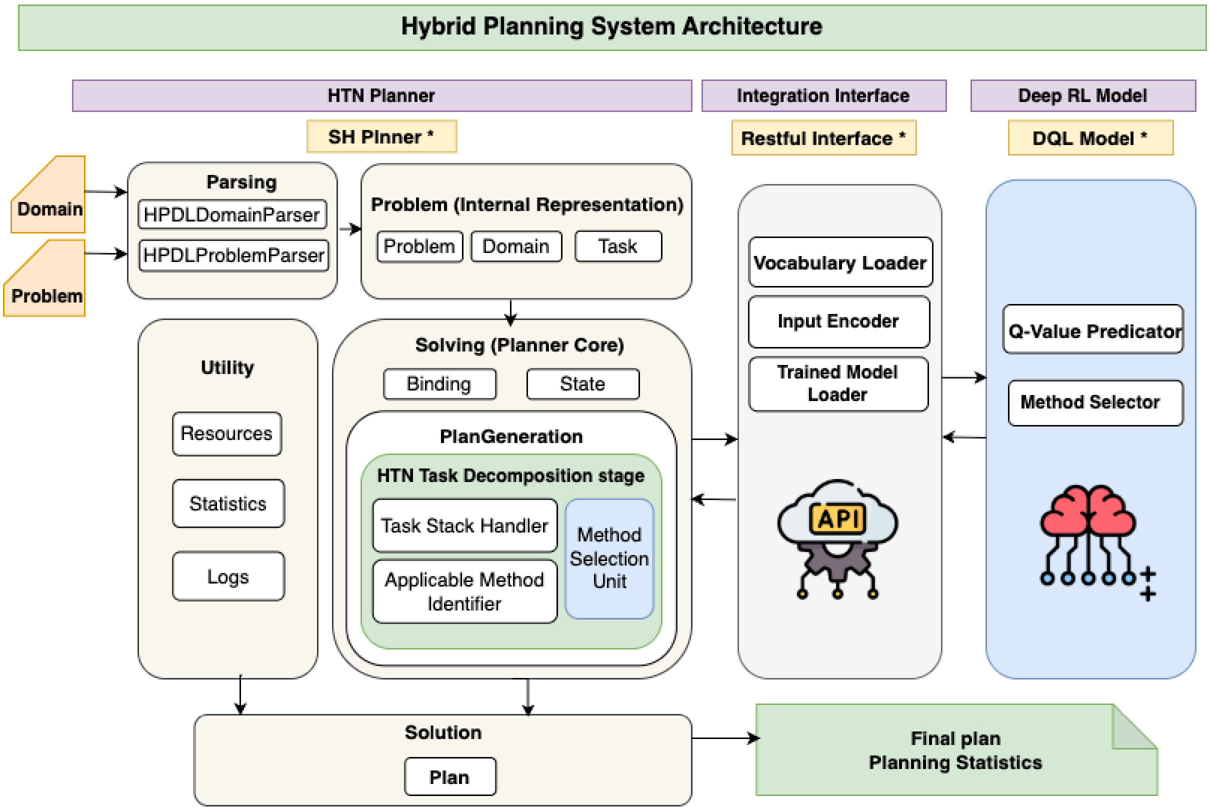


Figure 4.1: System architecture of the proposed hybrid planning framework, showing the interaction between the symbolic planner, integration interface, and learning model.[1]

4.2 Learning-Compatible Domain Principles

The design of an HTN domain plays a critical role in determining whether RL can be effectively applied for method selection. In particular, a learning-compatible domain must expose planning situations, during task decomposition, that present clear, repeatable decision points for the learning model to exploit. These

[1]Figure created by the author using diagrams.net. Icons were sourced from Flaticon.

decision points arise when multiple decomposition methods are available for a given task and differ meaningfully in their expected planning outcomes. Domains that support such structure are more likely to yield high-quality offline datasets that facilitate effective training of DRL models. To guide the assessment and adaptation of existing domains, we draw on a categorization of method alternativity proposed in [AG23; AGA22], which classifies methods as *strict alternatives*, *weak alternatives*, or *no alternatives* based on their precondition compatibility and simultaneous applicability.

This thesis focuses on tasks whose decomposition methods qualify as *strict alternatives*, or can be adapted to meet that criterion, as these offer the clearest learning opportunities. This principle informed the refinement of existing HTN domains to improve their suitability for training. Based on this perspective, we propose the following principles and characteristics for evaluating and adapting HTN domains to support learning-based method selection.

Principle 1: Favor Tasks with Strictly Alternative Methods. From a learning standpoint, it is essential that multiple methods for a compound task be applicable at the same decision point. Strict alternatives are particularly favorable because their applicability is determined solely by grounded state predicates, and their preconditions are mutually compatible. This ensures that the learning model regularly encounters real decision points during training. These principles guided the domain refinement process in this thesis.

- **Characteristic 1.1: Simultaneous Applicability.** The methods of a compound task should be applicable under the same or similar grounded states. If only one method is ever applicable, the agent has no meaningful choice to make. An example of a task from Blocksworld domain is shown in Figure 4.2, where two methods can be applied simultaneously without conflicting preconditions, illustrating the desired structure.
- **Characteristic 1.2: Non-Conflicting Preconditions.** Methods must not mutually exclude one another. If their preconditions cannot be satisfied together, they do not form strict alternatives and thus reduce learning opportunities. An example of this kind of task from Blocksworld domain is shown in Figure 4.3, where no decision point exists because methods never overlap in applicability.

Principle 2: Promote Structurally Diverse Outcomes. In addition to being applicable at the same time, the available methods should produce distinguishable downstream effects. This is important for generating reward signals that differentiate stronger strategies from weaker ones and allows the DRL model to learn an effective selection policy.

- **Characteristic 2.1: Distinct Reward Profiles.** The outcomes of alternative methods should differ measurably in terms of planning performance metrics such as plan length, time, or memory consumption.
- **Characteristic 2.2: Strategy Diversity.** Methods should implement distinct decomposition strategies (e.g., proactive manipulation vs. task recognition), enabling the model to learn under what conditions each is preferable.

Figure 4.2 illustrates an example from the Blocksworld domain, *do_on_table* task, where two methods are not only simultaneously applicable, but also differ significantly in both expected reward and planning strategy, one performing an environment-altering action sequence, the other recognizing that the task has already been achieved.

```
(:task do_on_table
    :parameters (?x - block)
   (:method m8_do_on_table ; if there is a block under block x
        :precondition (and (clear ?x) (handempty) (not (ontable ?x)) (block ?y) (on ?x ?y))
        :tasks (sequence (unstack ?x ?y) (put-down ?x))
   )
   (:method m7_do_on_table; Already satisfied
        :precondition (and (clear ?x))
        :tasks (sequence (bk-nop))
   )


)
```

Figure 4.2: The *do_on_table* task in the Blocksworld domain, with methods that are simultaneously applicable and produce structurally diverse outcomes.

```
 (:task do_clear
      :parameters (?x - block)
      (:method m12_do_clear
          :precondition (and (not (clear ?x)) (block ?y) (on ?y ?x) (handempty))
          :tasks (sequence (do_clear ?y) (unstack ?y ?x) (put-down ?y))
      )
      (:method m11_do_clear
          :precondition (and (clear ?x))
          :tasks (sequence (bk-nop))
      )
 )
)
```

Figure 4.3: The *do_clear* task, whose methods cannot be applicable at the same time due to the conflicting preconditions (red lines). These do not represent meaningful decision points and are excluded from training .

It is important to note that this assessment, and any necessary adaptation, does not need to be exhaustive. In practice, a domain does not need to satisfy the above principles and characteristics for every compound task. Rather, ensuring that a substantial portion, such as half or more, of the compound tasks support strict alternatives is typically sufficient to construct an action space rich enough to train a robust method selection policy.

4.3 Modeling the Planning Context for Method Selection

To enable the DRL model to serve as an intelligent method selector for HTN planning, it must first be trained on task decomposition data derived from the planning process. Specifically, the planning context must be abstracted into structured representations of states, actions, and rewards. This section outlines the key elements extracted from the HTN planning process to define the RL *state*, *action space*, and *reward function*. These abstractions are designed to capture general decision-making patterns while remaining compact and expressive enough to distinguish between alternative decomposition strategies. The representation design principles used to define the state, action, and reward are domain-independent and can be applied across different planning domains without modification.

4.3.1 State Representation

In this work, each decision point encountered during HTN task decomposition defines an RL state. It is important to distinguish this notion of state from the conventional HTN planning state, which refers to the set of grounded predicates representing the current world configuration. In a state-based HTN planner, the planning state only changes as a result of executing primitive actions, once a complete decomposition has been produced and applied. However, this transition frequency is too coarse for RL, where the agent requires finer-grained feedback to learn an effective policy.

To enable learning at the level of method selection, we redefine the RL state to correspond to each decomposition decision point. This means that multiple RL states may be observed within a single planning state, as the planner explores successive compound task decompositions. By capturing contextual information at each method selection step, the RL state allows the learning model to make informed choices that shape the structure and efficiency of the resulting plan. Table 4.1 summarizes the selected features used to construct the RL state. They were chosen to capture method selection context and support reward computation.

Feature Name	**Description**
current_task	Identifies the compound task currently being decomposed. It provides the decision context for selecting among applicable methods.
number_of_pendingTasks	Indicates the number of compound tasks that remain to be decomposed, reflecting structural progress within the plan.
UpdatedPendingTasks	Describes how the selection of each applicable method would affect the number of pending tasks, helping assess the decomposition impact of each option.
Applicable_methods_list	Specifies which decomposition methods are valid choices for the current task under the current planning state. This forms the set of actions available to the learning agent.
applicable_method_p_subtasks	Indicates how many primitive subtasks each applicable method introduces, giving insight into how close a method is to producing executable actions.
applicable_method_cp_subtasks	Indicates how many compound subtasks each applicable method introduces, indicating the level of abstraction or recursion introduced by the method.

Table 4.1: Key features used in the RL state representation

4.3.2 Action Space

In the context of task decomposition in HTN planning, the action space at each decision point consists of the set of decomposition methods applicable to the current compound task. These methods represent alternative ways to decompose the task into subtasks, and each applicable method is treated as a distinct action by the RL agent. The goal of the learning model is to select the most suitable method from this set, balancing decomposition efficiency and long-term planning performance. The set of applicable methods is determined by evaluating the precondition set of each method against the current planning state. Since different tasks, and even the same task in different contexts, may have varying numbers of applicable methods, the size of the action space changes across decision points. This dynamic and context-sensitive action space structure ensures that decision-making remains aligned with the hierarchical semantics of the domain, enabling learning models to reason about meaningful alternatives at each stage of decomposition.

4.3.3 Reward Function Design

The reward function plays a central role in shaping the behavior of the DRL model by quantifying the effectiveness of each method selection during task decomposition. This work adopts a composite reward that captures multiple dimensions of plan quality and structural efficiency. This multi-component design allows the model to receive more informative feedback from each decision point, even in the absence of complete plan success, which is especially important in offline training scenarios. Each component of the reward function reflects a specific planning objective, such as favoring simpler methods, reducing unnecessary recursion, or progressing toward the goal more efficiently. Table 4.2 summarizes the role of each reward and penalty component included in the formulation.

The reward $R(s, a)$ assigned to a method a in state s is computed as:

$$\begin{aligned} R(s,a) = \; & \overbrace{R_{\text{success}}}^{\text{Base Reward}} + \overbrace{\lambda_1 \cdot \frac{N_{\text{primitive}}(a)}{N_{\text{plan length}}}}^{\text{Simplicity Reward}} - \overbrace{\lambda_2 \cdot \frac{N_{\text{compound}}(a)}{N_{\text{total decomposition}}}}^{\text{Task Complexity Penalty}} \\ & + \underbrace{\lambda_3 \cdot \frac{N_{\text{pending}}(a)}{N_{\text{total decomposition}}}}_{\text{Goal Task Penalty}} - \underbrace{\lambda_4 \cdot \frac{N_{\text{bindings}}(a)}{N_{\text{total objects}}}}_{\text{Binding Complexity Penalty}} + \underbrace{R_{\text{bk-nop}} \cdot \delta_{\text{bk-nop}}(a)}_{\text{No-Op Bonus}} \end{aligned}$$

The indicator function $\delta_{\text{bk-nop}}(a)$ is defined as:

$$\delta_{\text{bk-nop}}(a) = \begin{cases} 1 & \text{if } a \text{ is a bk-nop/task() method,} \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

Reward/Penalty Component	Description
Base Reward	A constant reward granted for each successful task decomposition, serving as the foundation for additional reward adjustments.
Simplicity Reward	Encourages selecting methods that yield a higher ratio of primitive subtasks, favoring more direct and executable decomposition paths.
Task Complexity Penalty	Penalizes methods that introduce a large number of compound subtasks, which increase recursion and delay execution.
Goal Task Penalty	Penalizes methods that significantly increase the size of the pending task list, which may impede progress toward achieving the final goal.
Binding Complexity Penalty	Penalizes methods with more complex variable bindings, as these indicate higher contextual specificity or potential computational overhead.
No-Op Bonus	Assigns a bonus when a method resolves a task already satisfied in the current state (e.g., bk-nop), helping the model recognize and prefer no-operation cases that avoid redundant work.

Table 4.2: Description of reward and penalty components used in the reward function

4.4 Structured Dataset Design: Medallion Architecture

A foundational element of the proposed framework is the design of a structured, modular data pipeline that transforms planner-generated information into learning-ready inputs. Rather than relying on an ad hoc or tightly coupled data preparation strategy, the system adopts the Medallion Architecture, a principled, layered pattern that organizes dataset refinement into three semantically distinct stages: *Bronze*, *Silver*, and *Gold* [Datnd]. Each stage corresponds to a conceptual layer of data quality improvement, supporting modular experimentation and reproducible offline learning workflows.

The Medallion Architecture, originally introduced in data engineering for managing large-scale analytical pipelines [Str25], offers transferable benefits for RL pipelines. This is particularly valuable in offline settings, where clear separations between raw data, processed features, and model-ready inputs are essential. In this framework, the architecture ensures that symbolic planning traces are incrementally structured, filtered, and transformed in a way that preserves semantic fidelity while optimizing learning utility. Figure 4.4 illustrates the high-level data refinement process as applied in this work. Table 4.3 outlines the design intent and abstract responsibilities associated with each layer.

By applying this layered architecture, the proposed framework ensures that dataset preparation remains conceptually organized, semantically interpretable, and adaptable to future design variations. This modular approach facilitates the controlled evaluation of feature sets, reward abstractions, and task encodings without compromising the integrity of the underlying planning data.

[2] Figure created by the author using diagrams.net. Icons were sourced from Vecteezy.com and are used under the Vecteezy Free License, which permits usage with attribution.

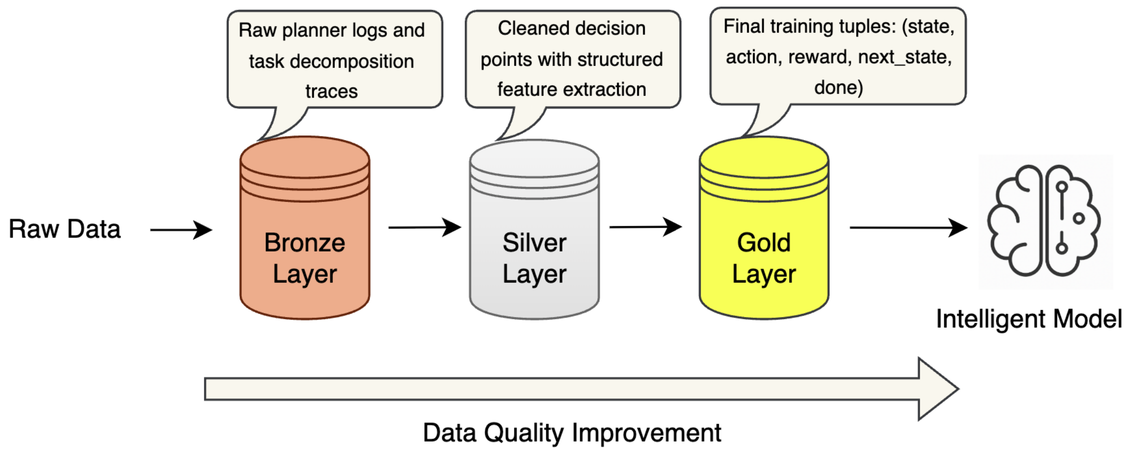


Figure 4.4: Adopted Medallion Architecture for the dataset pipeline. Each layer increases data quality and structure, preparing planner logs for RL.[2]

Layer	**Design Responsibilities**
Bronze	Serves as the raw data layer, capturing unprocessed outputs from the planner across diverse problem instances. This layer provides a comprehensive and traceable record of planning behavior without introducing filtering or abstraction.
Silver	Acts as the refinement layer, transforming raw traces into structured decision points. It supports semantic enrichment and removes uninformative or redundant data elements, enhancing signal quality for learning.
Gold	Represents the model-ready layer, organizing refined decision data into a format suitable for training learning models. This layer aligns the structured planning context with the decision modeling interface.

Table 4.3: Abstract responsibilities of Medallion Architecture layers in the dataset pipeline

5 Implementation

This chapter presents the implementation details of the proposed hybrid HTN planning framework, DQL-SH. Building upon the design concepts introduced in the previous chapter, this chapter focuses on the concrete realization of the system's core components. These include data generation, offline model training, Application Programming Interface (API) deployment, and the integration of the learning module into the SH planner. Each component corresponds directly to the architectural decisions outlined in Chapter 4, and is implemented using a modular and reproducible structure. Although this implementation is instantiated using the SH planner and a deep Q-network, the framework is designed to be independent of the specific HTN planner and DRL model employed, owing to its modular and decoupled architecture. Throughout the chapter, emphasis is placed on practical considerations, integration challenges, and the key engineering trade-offs encountered during development.

5.1 System Execution Instructions

To run the planning framework, both the baseline SH planner and the DQL planner can be executed locally with minimal setup, assuming all dependencies are installed. This section outlines how to compile and launch both variants of the system, as well as how to configure the planning inputs.

5.1.1 Running the Baseline SH Planner

To run the original SH planner in standalone mode, use the following steps in a terminal:

```
$ cd baseline-sh-planner/
$ sbt compile
$ sbt "runMain org.planx.sh.services.PlanningServices"
```

Upon execution, the planner begins the task decomposition process and prints the resulting plan and key planning statistics to the terminal. Additional logs, including planning time and method selections, are saved under:

```
$ baseline-sh-planner/logs/
```

To specify the planning configuration, such as the domain and problem file paths, one must modify the main method implemented inside *PlanningServices.scala*. This method was custom-developed to support easy batch testing and manual evaluation. The file paths should be set explicitly in this method before compilation and execution.

5.1.2 Running the DQL-SH Planner

To enable the DQL integration, both the SH planner and the FastAPI model service must be executed concurrently in two separate terminals.

Terminal 1: Launch the FastAPI Model Service Navigate to the API service directory and run:

```
$ cd FastAPI/
$ uvicorn FastAPI:app --reload
```

This starts the RESTful inference service, loads the trained DQL model, and begins listening for incoming HTTP requests at `localhost:8000`.

Terminal 2: Run the DQL-SH Planner In a second terminal, launch the SH planner as follows:

```
$ cd dql-sh-planner/
$ sbt compile
$ sbt "runMain org.planx.sh.services.PlanningServices"
```

The planner will now query the FastAPI service for method selection at each compound task during decomposition. Results are printed in the terminal and also logged to the usual output directory.

5.2 Technology Stack and Development Environment

The implementation of the proposed DQL-SH planning framework relies on a diverse set of technologies and tools, reflecting the interdisciplinary nature of the system, which spans conventional HTN planning, RL, API engineering, and software integration. This section outlines the key components of the technology stack, as well as the development environment used throughout the project.

5.2.1 Technologies and External Resources

Two core programming languages play distinct roles in the proposed system. The baseline SH planner is implemented in Scala and described in recent literature [GPAA24]. The codebase used in this work is obtained from the official GitHub repository[1] and is reused without modification to its core planning logic. Minor extensions were made to extract internal planning variables and insert logging functionalities required for dataset generation. Scala is also used to interface with external components via Akka HTTP, enabling communication with the learning-based components.

All learning-based components of the framework, including the data processing pipeline, DQL model, and REST API service, are implemented in Python. The rich machine learning ecosystem of Python, including libraries such as PyTorch, NumPy, Pandas, Matplotlib, and FastAPI, supports efficient model development, training, and deployment.

The raw planning problems used to generate both training and testing data belong to the Blocksworld domain, a classical AI planning benchmark, and are sourced from the publicly available HTN-MAKER dataset[2] developed by [HKM10].These 800 PDDL problem files are adapted into HPDL format for compatibility with the SH planner, as detailed in Subsection 5.3.2. The choice of Blocksworld as the foundation for learning-based HTN planning is further justified in *Domain Adaptation for Learning Compatibility* (Section 5.3.1).

[1] https://github.com/PlanX-Universe/sh
[2] https://www.cse.lehigh.edu/InSyTe/HTN-MAKER/

5.2.2 Development Environment and System Configuration

The implementation is carried out on a macOS-based system using a hybrid software stack involving both Python and Scala. Table 5.1 summarizes the hardware and software configurations used throughout the project.

Component	Specification
Operating System	macOS Sequoia (Version 15.0, Build 24A335)
CPU	Apple M1
Memory	16 GB RAM
Java/Scala Environment	OpenJDK 17 and Scala 2.13, installed in the base system environment
Python Environment	Python 3.10 with PyTorch 2.0, managed in an isolated Conda environment
Development Tools	Visual Studio Code (VSCode) with Metals plugin for Scala and Python extension for Python
Version Control	Git, with repositories hosted on GitHub

Table 5.1: Hardware and software configuration of the development environment

5.3 Data Creation and Preprocessing

This section outlines the process of generating and transforming data used to train the DQL model. As introduced in earlier chapters, the offline learning setup adopted in this thesis requires structured training data composed of trajectories that capture state, action, and reward information at each decision point during planning. To support this requirement, the data creation pipeline extracts annotated trajectories from problems solved by the baseline SH planner. Before initiating the data collection process, a dedicated domain adaptation step ensures that the planning environment exhibits structural properties conducive to learning-based method selection. This adaptation phase is described in detail in Subsection 5.3.1. To ensure reproducibility, modularity, and clarity, the system adopts a medallion-style data architecture that separates transformation steps into distinct layers. The following subsections describe this pipeline in detail, including planner execution, trace logging, and the progressive refinement of decision-point data across the Bronze, Silver, and Gold layers. In addition to training data generation, the same environment is also used to prepare a held-out test set, which is converted into HPDL format and later executed by both planners for performance evaluation.

5.3.1 Domain Adaptation for Learning Compatibility

Since classical planning domains are not inherently designed for learning purposes, they might lack one or both of the core principles, *strict alternatives* and *structurally diverse outcomes*, or fail to satisfy their key characteristics, as discussed in Section 4.2 (*Learning-Compatible Domain Principles*). To avoid this situation, it is useful to systematically assess and, where necessary, enrich the domain to improve its learning compatibility. A practical strategy involves identifying compound tasks with limited method diversity and designing additional, progressively smarter decomposition methods. This can be achieved by adding more specific predicates to the preconditions of new methods, making them applicable in more constrained contexts, and by reducing the number of required subtasks. In doing so, each method offers a distinct trade-off between effort and generality while still achieving the same compound task.

We use Blocksworld domain as the primary environment for this thesis due to its widespread recognition as a classical benchmark in AI planning. Its clear object manipulation rules, intuitive semantics, and support for hierarchical task decomposition make it an ideal setting for exploring method selection in HTN planning. Beyond its simplicity and interpretability, Blocksworld is particularly well-suited for learning-based approaches: nearly all of its compound tasks feature decomposition methods that qualify as strict alternatives, as defined in Section 4.2. This ensures the presence of frequent and meaningful decision points for the learning model, while also enabling structural refinements that align with learning-compatible design principles. As such, Blocksworld provides both a controlled experimental setup and sufficient variability in method choice to effectively evaluate the proposed DRL-enhanced planning framework.

While the structure of this domain is largely compatible with the goals of learning-based method selection, certain tasks present opportunities for further enhancement. A representative example is the *do_put_on* task, shown in Figure 5.1. One of its methods, *m1_do_put_on*, is already classified as a strict alternative due to its compatibility with other applicable method. However, its task list is particularly well-structured for additional refinement: it consists of a linear sequence of subtasks, *do_clear*, *do_on_table*, and *do_move*, and notably excludes any recursive call to the parent task. This makes it structurally safe and semantically meaningful to break *m1_do_put_on* into multiple, more specific methods that impose stricter preconditions while containing fewer subtasks in their task lists. Each such method reflects a planning state in which some of the subtasks from the original, more general method, have already been achieved, thereby allowing for more efficient and context-sensitive decompositions.

This decomposition strategy aligns with both Principle 1 (strict alternatives) and Principle 2 (structural diversity), enabling the construction of method alternatives that are simultaneously applicable and yield distinguishable planning outcomes. By enriching the domain in this manner, we preserve its original strengths while amplifying its value as a learning environment, ultimately producing richer decision points and more informative offline training signals for the DQL model.

Figure 5.2 visualizes the result of this enrichment process, showcasing five distinct methods introduced for the *do_put_on* task. Each method builds on the original decomposition structure while selectively reducing the number of subtasks based on stricter preconditions. The figure makes it apparent how the original method was modularized into smarter alternatives, each applicable under a specific planning context. These additions not only expand the action space available to the learning agent but also illustrate the practical implementation of both strict alternativity and structural diversity within the same task. As such, the figure serves as a concrete example of how theoretical domain principles were translated into actionable domain refinements during implementation.

```
(:task do_put_on
    :parameters (?x - block ?y - block)
    (:method m0_do_put_on
        :precondition (and (on ?x ?y))
        :tasks (sequence (bk-nop))
    )
    (:method m1_do_put_on
        :precondition (and (handempty))
        :tasks (sequence (do_clear ?x) (do_clear ?y) (do_on_table ?y) (do_move ?x ?y))
    )
)
```

Figure 5.1: Original *do_put_on* task in the Blocksworld domain.

```
(:task do_put_on
 :parameters (?x - block ?y - block)
    (:method m6_do_put_on  ; Clear both then move
        :precondition (handempty)
        :tasks (sequence (do_clear ?x)
                         (do_clear ?y)
                         (do_on_table ?y)
                         (do_move ?x ?y))
    )
    (:method m5_do_put_on  ; When ?y is already clear
        :precondition (and (not (clear ?x))(clear ?y) (handempty))
        :tasks (sequence (do_clear ?x)
                         (do_on_table ?y)
                         (do_move ?x ?y)))

    (:method m4_do_put_on  ; When ?x is already clear
        :precondition (and (not (clear ?y))(clear ?x) (handempty))
        :tasks (sequence (do_clear ?y)
                         (do_on_table ?y)
                         (do_move ?x ?y))
    )
    (:method m3_do_put_on  ; both objects are clear
        :precondition (and (not (clear ?y))(clear ?x)(ontable ?y)(handempty))
        :tasks (sequence (do_clear ?y)
                         (do_move ?x ?y))
    )
    (:method m2_do_put_on  ; both objects are clear
        :precondition (and (not (clear ?x))(clear ?y)(ontable ?y)(handempty))
        :tasks (sequence (do_clear ?x)
                     (do_move ?x ?y))
    )
  (:method m1_do_put_on  ; both objects are clear
    :precondition (and (clear ?x)(clear ?y) (handempty))
    :tasks (sequence (do_on_table ?y)
                     (do_move ?x ?y))
 )

 (:method m0_do_put_on  ; Already satisfied
    :precondition (on ?x ?y)
    :tasks (sequence (bk-nop))
 )
```

Figure 5.2: Enriched *do_put_on* task. The enriched version presented in this thesis includes additional methods with more restrictive preconditions and fewer required subtasks. These methods are simultaneously applicable in overlapping planning states, creating richer decision points and enabling more informative learning signals.

5.3.2 Raw Problem Conversion

A set of 800 raw planning problems in PDDL format belonging to the Blocksworld domain serves as the input dataset. To ensure compatibility with the baseline SH planner, these problem files must first be converted into HPDL format. The conversion process is managed in the `problem_files` folder within the baseline SH planner directory, which includes the raw problem files under *input_raw_problem_files*, as well as a configuration file in JSON format.

This configuration file is used to align naming conventions, such as predicate and task names, with those defined in the domain. In this thesis, the domain file is taken directly from the benchmark domain folder included with the SH planner. However, as previously mentioned, the raw Blocksworld problem files are obtained from a separate benchmark set, which uses slightly different naming conventions. As a result, certain predicates must be renamed to ensure compatibility. For example, terms such as *hand-empty* and *on-table* in the raw PDDL files are renamed to *handempty* and *ontable*, respectively, to match the domain specification of the SH planner.

Two Python scripts support this conversion process. The first script, called the converter, structurally transforms files with PDDL format into HPDL by adapting its syntax, naming, and goal specification. The second script, referred to as the extender, ensures that each problem instance includes a sufficiently rich goal set. Specifically, in the Blocksworld domain used in this thesis, if the number of defined `do_put_on` tasks is too small relative to the number of objects in the problem, the script automatically appends additional tasks. This is done by identifying unused blocks in the current goal structure and chaining them into new `do_put_on` tasks, using the last used block as the stacking base. This extension process increases both the structural complexity and the variety of the resulting problems, thereby producing richer training and testing data. Figure 5.3 presents excerpts from a raw PDDL problem and its converted HPDL equivalent, highlighting key structural differences.

<table>
<tr><th>Original PDDL format</th><th>Converted HPDL Format</th></tr>
<tr><td><pre>...
(:init
  (on-table b006)
  (clear b006)
  ...
  (hand-empty)
)
(:goal
  (and
    (on b003 b002)
    (on b007 b003)
    ...
  ))</pre></td><td><pre>...
(:init
  (ontable b006)
  (clear b006)
  ...
  (handempty)
)
(:goal-tasks (sequence
  (do_put_on b003 b002)
  (do_put_on b007 b003)
  ...
))</pre></td></tr>
</table>

Figure 5.3: Comparison between original PDDL and converted HPDL problem formats

5.3.3 Planner Execution and Logging

After converting the raw problem files into HPDL format, the dataset of 800 problems is partitioned into two subsets: 700 problems are used to generate training data, and the remaining 100 are reserved for testing. For each problem, the SH planner is executed to produce decision-point logs that capture relevant decomposition and planning details during execution. These logs serve as the basis for constructing (state, action, reward) trajectories used in offline DQL training and evaluation.

To facilitate structured data collection and runtime monitoring, the SH planner is instrumented with a lightweight but extensible logging infrastructure. Logging is configured using the SLF4J API in combination with the Logback framework, which is defined in the *logback.xml* configuration file located at *resources* package of both the baseline and DQL-SH planner directories. This setup employs a dual logging strategy: a console appender reports messages at the `INFO` level or higher, while a file appender captures detailed *DEBUG*-level outputs in a log file named according to the pattern *domainName_problemName.log*, reflecting the domain and problem instance being solved.

The core of the data collection mechanism is a global utility object named *Loggs*, implemented in the *utility* package. This object contains scalar fields representing the internal state of the planner at each decision point, including task decomposition details, selected method, applicable method sets, and intermediate planning variables. Each field is semantically aligned with a decision component described in Subsection 4.3.1 of Chapter 4. During execution, values are assigned to these fields across multiple modules, particularly within *PlanGeneration.scala* and *Task.scala*, where the core of the planning process occurs.

At each decision point, the planner captures a snapshot of the current state by logging the relevant fields through a *DEBUG* block. This block is executed only if logging is enabled at the debug level, allowing performance-efficient toggling for large-scale runs. After logging, the global state is reset via a *Loggs.reset ()* call to prepare for the next decision point. Additionally, a call to *Loggs.nextState ()* encodes the next decision point in a format compatible with the DRL model, specifically representing the next RL state as defined in chapter 4.

The resulting *domainName_problemName.log* file contains a structured sequence of decision-point dictionaries, each corresponding to a planning choice made by the SH planner. These logs form the foundation for offline data generation, enabling the extraction of (state, action, reward) trajectories for DQL.

5.3.4 Medallion Architecture Implementation

As introduced in Chapter 4, the system adopts a Medallion Architecture to structure the data preparation pipeline into distinct and progressively enriched layers. This implementation follows the same layered structure to ensure a traceable, modular, and extensible data flow from raw planner logs to learning-compatible training examples. The datasets and scripts corresponding to each layer are organized within the *RL_framework* directory: three subfolders, *1-Bronze*, *2-Silver*, and *3-Gold*, contain the Python scripts responsible for each transformation stage, while the *0-dataset* folder stores the resulting CSV files for all layers. The following paragraphs describe the logic and implementation details of each stage in the pipeline.

Bronze Layer

The Bronze layer serves as the entry point for transforming raw planning logs into structured datasets. It processes the unstructured log files generated during planner execution and converts them into a standardized tabular format. A custom Python parser is used to read each log file, extract decision point blocks, parse nested fields, and record relevant planning metrics. The parser robustly handles various value formats, including function calls, comma-separated strings, numeric values, and list notations, ensuring accurate and consistent data representation. For each decision point, the parser extracts the current state and action details, along with metadata such as episode ID, and filename. Each planning problem is treated as a separate RL episode, and all log records corresponding to that problem share the same episode ID. This ensures a clear mapping between individual problem files and their associated decision-point records within the dataset. The parser batch-processes all log files in the dataset directory and consolidates the results into a single CSV file comprising **15,979** records, which forms the raw but structured output of the Bronze layer. This output is stored as *blocksworld_bronze.csv* and serves as the foundation for further processing in the Silver layer.

Silver Layer

Serving as a bridge between raw log data and learning-ready inputs, the Silver layer transforms the unfiltered Bronze dataset into a concise, well-structured intermediate representation. At this stage, the raw *Bronze.csv* is parsed and standardized: all columns containing list or dictionary structures are converted into structured Python objects, and five key fields, including *Applicable_methods_list*, *UpdatedPendingTasks*, *applicable_method_p_subtasks*, *applicable_method_cp_subtasks*, and *numberOfCandidates* are cleaned and standardized to ensure numerical accuracy and categorical consistency. The main task name is extracted and standardized to serve as a reliable RL state feature.

Following this, advanced feature engineering is performed: a *delta* indicator is calculated for each method, taking the value 1 whenever both primitive and compound subtask counts are zero, and 0 otherwise; this indicator is incorporated in the reward function as a dedicated *no-op bonus*. The reward for each decision point is then computed according to the explicit formula presented in chapter 4 (Reward Function subsection 4.3.3), which consists of a base reward for success, a simplicity reward for primitive actions (normalized by plan length), a task complexity penalty for compound decompositions (normalized by the number of evaluated states), a goal task penalty for pending tasks, a binding complexity penalty for overloaded candidate bindings, and a no-op bonus for delta-flagged methods. All penalties and bonuses are normalized as specified to ensure the reward signal is robust and comparable across problems of varying complexity.

To enhance data quality and learning effectiveness, rows where only one method is applicable are removed, so that the dataset focuses exclusively on genuine multi-choice decision points. Non-essential metadata and planning statistics are also dropped, and only a concise set of RL-relevant columns is retained. The result is a compact, intermediate dataset, *Silver.csv*, containing **6,249** records, each representing a structured decision point suitable for subsequent conversion to final (state, action, reward) tuples in the Gold layer.

Gold Layer

The Gold layer performs the final transformation of the Silver dataset into a training-ready format suitable for DQL, bridging the gap between SH planner output and DRL model input. This stage begins by constructing compact vocabularies for all unique tasks and methods present in the data, assigning each a unique numeric identifier and storing the resulting mappings in JSON files for transparency and reproducibility. Raw string-encoded lists in the *Silver.csv* are systematically converted to Python objects, after which synchronized shuffling (i.e., preserving index alignment across fields) is applied to each decision point's applicable methods, pending tasks, subtask counts, and reward values. This shuffling is necessary because, in the original planner logs, the planner's default behavior is to select the first applicable method in the applicable methods list; without shuffling, the learning model would pick up this positional bias and always favor the first method, rather than learning from features of the planning state or method itself. By randomizing the order while preserving the correspondence between methods and their associated rewards, the model is encouraged to develop a genuine policy based on task context, not list order.

To support neural network architectures with fixed input dimensions, the Gold layer pads all variable-length state and action lists to a uniform length using a designated padding value (-10.0), while numeric features (such as subtask counts and pending task statistics) are normalized on a per episode basis to the $[0, 1]$ range, guaranteeing comparability across planning instances of different complexity. Each planning state is then represented as a dictionary that includes the encoded ID of the current task, the number of pending tasks, the list of applicable method IDs, and all subtask-related metrics, with numerical values normalized.

A distinctive aspect of this layer is the expansion of each row into multiple training examples: for every decision point with M applicable methods, the Gold layer generates M separate (state, action, reward, next_state, terminated) tuples, providing explicit coverage of all state-action-reward options encountered during planning. This expansion ensures that all possible actions encountered during data collection are represented in the training dataset. The explicit *terminated* flag marks episode boundaries, and next-state padding ensures that transitions at the end of each episode are handled consistently.

The result is stored in *Gold.csv*, comprising **12,948** records, where each record represents a complete transition containing the encoded planner state, the selected action, the shaped reward, the resulting next state, and an episode termination marker. Through these procedures, advanced encoding, normalization, synchronized shuffling, and state-action expansion, the Gold layer yields a dataset fully optimized for efficient and unbiased offline DQL training, directly supporting batch learning and robust generalization to diverse planning scenarios.

5.4 Deep Q-Learning Model: Architecture and Training Strategy

This section presents the concrete implementation and training procedure for the DQL model integrated into the SH planning framework. Leveraging the Gold dataset of structured (state, action, reward) transitions, the model is constructed to approximate optimal method selection policies for the planner. The following subsections document the final network architecture, the training workflow, hyperparameter choices, and explain the validation and early stopping strategies used to ensure effective and robust model performance.

5.4.1 Offline Training Pipeline

The offline DQL model is trained through a structured workflow that transforms planner traces into a learned method selection policy. This training process is organized into several sequential stages that handle data preparation, model inference, and policy learning. As illustrated in Figure 5.5, the pipeline begins with pre-collected traces from the SH planner, which are processed into a Gold dataset. These transitions are used to train a DNN that approximates the Q-function, a mapping from planning decision point states to the expected utility of each applicable method. Each stage of this training pipeline is described in detail below:

Offline Transition Data and Data Loader. As mentioned before, the offline learning setup relies on a dataset of pre-collected planning transitions. Each transition is stored as a tuple of the form $(s_t, a_t, r_t, s_{t+1}, \text{done}_t)$, where s_t represents the state before method selection, a_t is the selected method, r_t is the scalar reward received, s_{t+1} is the resulting state, and done_t indicates whether the episode has terminated. These transitions are stored in the Gold dataset and loaded into memory using a PyTorch-based data pipeline. A custom *Dataset* class is implemented to parse and preprocess each structured record into tensor-based inputs, while the *DataLoader* module handles batching and data access during model training.

Mini-Batch Sampling and Input Flow. The training loop operates on fixed-size mini-batches generated by the *DataLoader*. Each mini-batch contains a randomized subset of transitions sampled from the Gold dataset, consisting of multiple $(s, a, r, s', \text{done})$ tuples. Each mini-batch serves as one atomic training step: it is passed through the Q-network for forward inference, loss computation using Bellman targets, and gradient-based parameter updates. By re-sampling and shuffling at every epoch, this mini-batch pipeline promotes generalization and reduces the risk of overfitting.

DNN Architecture Implementation. The neural network used in this work follows a straightforward yet effective architecture tailored for offline DQL in the context of method selection. It is composed of an input layer that receives the flattened, preprocessed planner state, where all variable-length attributes are padded using a designated value (-10.0) and normalized into the range $[-10, 1]$ or $[0, 1]$, depending on the feature type. This is followed by two hidden layers and an output layer. Each hidden layer contains 64 units, where every unit is connected to all units in the previous layer, a structure known as a fully connected or dense layer. After each of these layers, a non-linear activation function called the *Rectified Linear Unit (ReLU)* is applied to introduce non-linearity into the model, allowing it to capture more complex patterns in the input data. The final output layer is a linear transformation that maps the last hidden layer's output to a vector of Q-values for each candidate method available at a decision point. In the Blocksworld domain, this corresponds to nine possible methods, resulting in nine output values. These Q-values estimate the expected future reward for selecting each method given the current planner state. Figure 5.4 illustrates the end-to-end flow of data through the model: from normalized input features, through the two fully connected layers and ReLU activations, to the output layer that produces the Q-values. This architecture is deliberately designed to balance expressiveness and generalization. It provides enough capacity to model dependencies between planning state attributes and method selection decisions, while remaining compact to reduce the risk of overfitting to the offline dataset. Moreover, the relatively small size of the network helps ensure that gradients, used during training to adjust the network's weights, remain stable and do not explode or vanish, a common concern in deeper architectures.

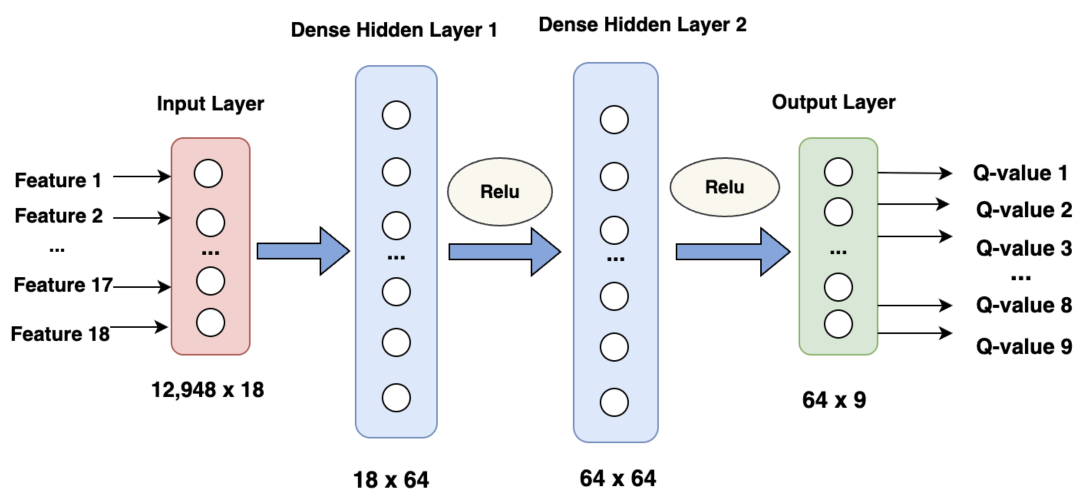


Figure 5.4: Schematic of the DQL network architecture. This figure is inspired by the general neural network architecture presented in [AG24].[3]

[3]Figure created by the author using diagrams.net.

Target Network. To improve learning stability, a second neural network, called the target network, is used to compute the target Q-values. This network has the same architecture as the main (policy) network but is updated less frequently. Specifically, while the policy network's parameters (θ) are updated after every mini-batch, the parameters (θ^-) of the target network are only periodically copied from the policy network. This mechanism reduces feedback loops where the model tries to chase its own changing predictions, thus stabilizing learning and helping the model converge toward more reliable Q-value estimates. During training, the target network is used to compute the Q-learning targets based on the Bellman equation [MKS+15]:

$$y = r_t + \gamma \cdot \max_{a'} Q_{\text{target}}(s_{t+1}, a'; \theta^-) \tag{5.1}$$

This equation states that the ideal target value y is the immediate reward r_t plus the discounted estimate of future reward, obtained by selecting the action a' with the highest Q-value in the next state s_{t+1} according to the target network Q_{target}.

Loss Function. The loss function in DQL quantifies the deviation of current predictions of the Q-network from a target value that represents a better estimate of the true expected return. This target, denoted by y, is computed as described in Equation 5.1, using the immediate reward and the maximum predicted Q-value of the next state according to the target network. The goal of the Q-network is to minimize the discrepancy between its prediction and this target value by optimizing the Mean Squared Error (MSE) [MKS+15]:

$$\mathcal{L}_{\text{MSE}} = (Q(s_t, a_t; \theta) - y)^2 \tag{5.2}$$

Optimizer. The network is trained using the *Adam* optimizer with a learning rate of 1×10^{-6}. Adam is well-suited to this setting due to its adaptive learning rate and momentum mechanisms, which support stable convergence even in noisy or imbalanced data scenarios. The optimizer is applied to minimize the loss defined in Equation 5.2 by updating the parameters θ of the main Q-network.

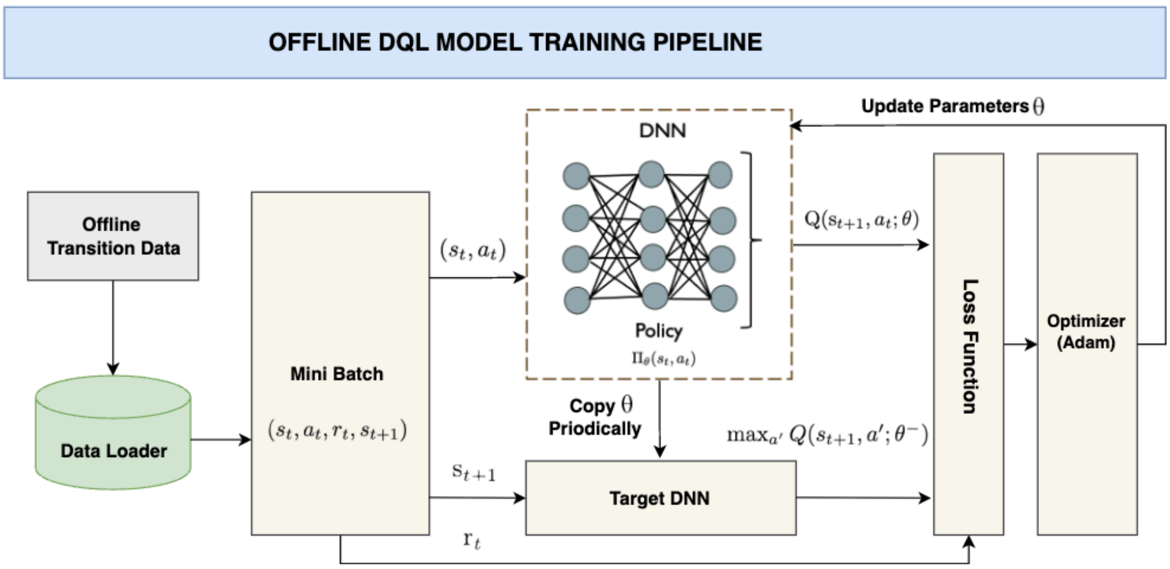


Figure 5.5: Offline DQL model training pipeline. This figure is inspired by the general DQL framework presented in [GVAM20].[4]

5.4.2 Optimization Techniques and Class Imbalance Handling

The training setup incorporates several stabilization strategies previously described in the pipeline, including mini-batch sampling, target network updates, and early stopping. Early stopping is applied with a patience of five epochs on a 10% validation split to prevent overfitting and ensure generalization. In addition, two numerical safeguards are used to improve robustness. First, gradient clipping with a maximum norm of 1.0 prevents exploding gradients during backpropagation. Second, reward values are clipped to the $[0, 1]$ interval and scaled by a factor of 3.0, ensuring that large rewards do not dominate training while preserving the learning signal from smaller values.

Handling Class Imbalance. One of the key modeling challenges in this domain is the natural class imbalance in method selection outcomes. This imbalance arises from structural properties of the planning benchmark domain: methods with more general preconditions are applicable in many planning states and therefore occur frequently, while more constrained methods appear less frequent due to their restrictive applicability. If unaddressed, this imbalance can bias the DQL model toward frequently seen actions and limit its ability to generalize across diverse planning situations.

To mitigate this issue, we adopt a data-level sampling strategy inspired by the general concept of *class-aware sampling techniques* [JK19]. Specifically, we use PyTorch's built-in `WeightedRandomSampler` [PyT24] to upsample underrepresented methods during training. Sampling weights are assigned based on the squared inverse frequency of each method, thereby amplifying the contribution of less frequent actions during optimization. For each training instance with action a_i, the sampling weight is computed as:

$$w(a_i) = \left(\frac{N}{n_{a_i} + \epsilon} \right)^2 \tag{5.3}$$

where N is the total number of samples, n_{a_i} is the number of times action a_i appears in the dataset, and ϵ is a small constant added for numerical stability. This weighting scheme ensures that less frequent but meaningful actions are sufficiently represented in training batches, enabling the model to learn a more balanced and robust policy over the entire method space.

5.4.3 Training Dynamics and Configuration Summary

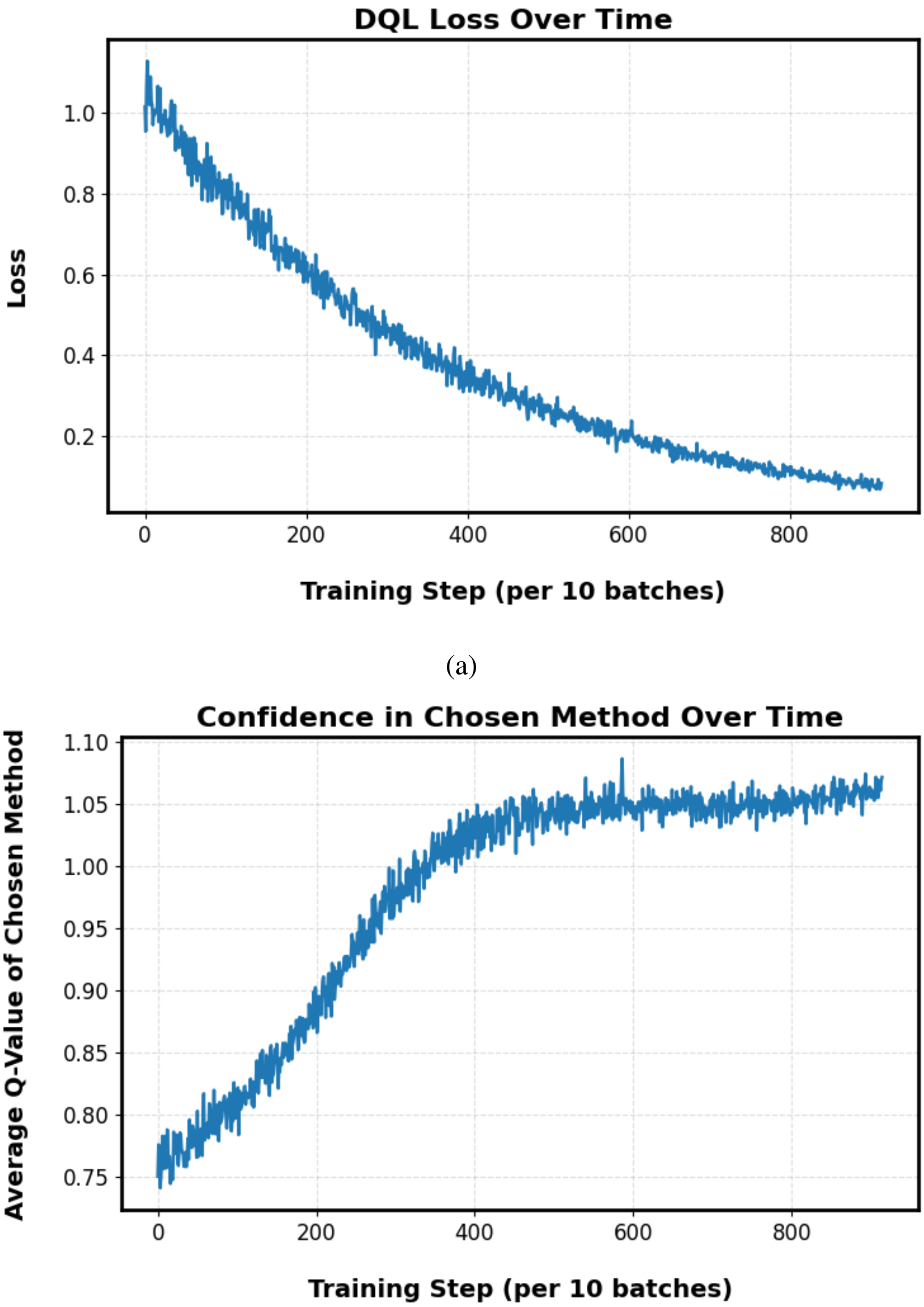
To support transparency and reproducibility, all hyperparameters related to model optimization, gradient control, early stopping, and reward shaping are summarized in Tables 5.2 and 5.3. These tables provide an overview of key learning configurations.

Figure 5.6 visualizes the training dynamics of the DQL model using two diagnostic curves. Subfigure *(a)* shows the smoothed loss over time, reflecting the model's progressive reduction of Q-value prediction error as training proceeds. The steady downward trend suggests effective convergence with no signs of overfitting or divergence. Subfigure *(b)* presents the average Q-value assigned to the chosen action in each mini-batch. This metric serves as an indicator of the model's growing confidence in its decisions. As training advances, the Q-values assigned to the selected methods increase and eventually stabilize, highlighting the model's ability to consistently identify high-reward actions across planning states.

In both plots, the x-axis shows training progress in units of 10 mini-batches, not epochs. This design choice stems from the smoothing strategy used during plotting, where every data point represents the average value over 10 consecutive mini-batches. Given that each mini-batch contains 64 training samples and the

[4] Figure created by the author using diagrams.net.

training set consists of roughly 11,653 samples, one full epoch spans approximately 182 mini-batches. As a result, an x-value of 200 corresponds to around 2,000 processed batches, which equates to roughly 11 training epochs. This batching-based x-axis offers a finer-grained view of learning dynamics and reveals trends that may be less visible with epoch-level granularity.



(a)

(b)

Figure 5.6: Visualization of the DQL training dynamics. The loss curve in (a) reflects decreasing Q-value prediction error, while the Q-value confidence in (b) indicates the model's increasing preference for high-reward methods.

Parameter	Value	Purpose
Learning Rate	1×10^{-6}	Prevents Q-value divergence
Batch Size	64	Balances noise and compute efficiency
Discount Factor (γ)	0.99	Encourages long-term reward consideration
Reward Scaling Factor	3.0	Amplifies learning signal
Gradient Clipping	1.0	Prevents instability
Validation Split	0.1	Supports early stopping
Early Stopping Patience	5 epochs	Avoids overfitting
Target Network Update	Every 2 epochs	Stabilizes Q-target computation
Loss Function	MSE	Measures Q-value error

Table 5.2: Training hyperparameters and configuration settings for the DQL model

Reward Component	Weight	Interpretation
Base Reward	+1.0	Base reward for progressing decomposition
Primitive Subtasks	+1.0	Simplicity incentive (scaled by plan length)
Compound Subtasks	−1.5	Penalizes complexity (scaled by evaluated states)
Pending Tasks	−2.0	Penalizes task overload
Binding Overload	−1.0	Penalizes method candidate explosion
Bonus Reward	+10.0	Strong bonus for safe undo / no-op moves

Table 5.3: Reward function coefficients used to shape the training signal.

5.5 REST API Integration for Planner Interaction

To enable real-time decision making during planning, this work introduces a RESTful API that serves as a communication bridge between the SH and a trained DQL model. This modular interface allows the planner to outsource method selection to a learned policy without modifying the core logic or architecture of the SH planner. FastAPI, a modern Python framework for building high-performance APIs, is used to implement the service, ensuring both speed and ease of integration.

5.5.1 API Service Architecture

The FastAPI service is designed to accept decision point state representations as input and return the most suitable decomposition method as determined by the trained DQN model. It exposes two endpoints, each configured for a distinct input format. The primary endpoint, *predict*, accepts structured JSON inputs that follow a predefined schema to ensure validation and data consistency. For greater flexibility, a secondary endpoint, *predict-raw*, allows input in plain text format. This raw string is parsed and converted internally to match the expected structure.

Internally, the service uses PyTorch for model inference, with a preloaded model that maps decision point states to Q-values over possible methods. *Uvicorn*, an *ASGI* server[5], is used to run the FastAPI service, enabling high-performance asynchronous request handling. This architecture allows the API to function efficiently in both development and production environments.

5.5.2 System Integration and Runtime Inference

The trained DQN model is deployed as a FastAPI service and used to support method selection during planning. The model architecture, input data configuration, and preprocessing steps are identical to those described earlier during training (see subsection 5.4.1). This ensures full compatibility between the planner's decision point states and the expected input format of the deployed model.

At runtime, the SH planner, implemented in Scala, interacts with the FastAPI service through a dedicated module called *DQLServiceCaller*. This client serializes each decision point state into a JSON object and sends it as a POST request to the FastAPI predict endpoint. The API processes the input, performs inference using the loaded model, and returns the name of the selected decomposition method. This method is then injected into the ongoing task decomposition process by the Method Selection Unit within the SH task decomposition engine.

To ensure that planner constraints are respected, Q-values corresponding to inapplicable methods are masked before final selection. The model then returns the applicable method with the highest predicted Q-value. The selected method is parsed from the API response and injected into the ongoing task decomposition process by the Method Selection Unit within the SH task decomposition engine.

This interaction occurs synchronously within the main planning loop defined in the *PlanGeneration* module. The planner blocks until a response is received to ensure determinism and maintain planning correctness. For evaluation purposes, the time spent waiting for API responses is measured separately from the overall planning time and excluded from the reported performance statistics.

5.5.3 Error Handling and Timeout Configuration

To ensure reliability during real-time interaction between the SH planner and the FastAPI service, robust error handling mechanisms are implemented on the Scala side. The *DQLServiceCaller* module includes a configurable timeout threshold and a limited retry mechanism to handle cases where the API service is temporarily unresponsive or experiences degraded performance. If the FastAPI server fails to respond within the specified timeout duration, or if a network error occurs, the client handles such exceptions gracefully, ensuring uninterrupted planning. In such scenarios, the system reverts to a predefined fallback strategy based on default method selection. This ensures that planning can continue in a degraded but functional mode, thereby preserving the overall responsiveness and fault-tolerance of the planner. Furthermore, detailed logging is employed to capture any communication failures, enabling easier debugging and post-run analysis. This setup prioritizes robustness in real-world deployments, where temporary service outages, network variability, or computational bottlenecks may occur. By decoupling the planning logic from strict model availability, the architecture prevents single points of failure and guarantees that critical planning processes are not halted due to downstream API issues.

[5] ASGI (Asynchronous Server Gateway Interface) is a standard for asynchronous Python web applications, enabling non-blocking request handling.

6 Evaluation

This chapter presents a comprehensive evaluation of the proposed DQL-enhanced SH planner (DQL-SH) against the baseline SH planner. The goal is to assess whether integrating a learning-based method selection strategy leads to measurable improvements in planning efficiency, solution quality, and resource usage. To ensure a transparent and reproducible comparison, the evaluation is divided into two main parts. First, Section 6.1 outlines the experimental design and setup, including the planners under comparison, the benchmark problem set, the selected evaluation metrics, and the execution environment. Then, in the subsequent sections, we analyze the results through descriptive statistics, visual comparisons, significance testing, and a qualitative case study. This chapter structure facilitates a logical progression from experimental design to empirical analysis, supporting a clear and systematic comparison of the planners' performance.

6.1 Experimental Setup

This section outlines the design of the evaluation process and the experimental conditions under which both planners are compared. The goal is to create a controlled and diverse environment to assess the effectiveness of learning-based method selection in HTN. We compare two planning systems: the *baseline SH* and the *DQL-SH planner*. Both are evaluated over a shared benchmark set consisting of 100 problem instances from the *Blocksworld* domain. These problems are generated to capture a broad spectrum of planning challenges. In particular, they vary in both the number of objects and goal tasks required per instance. This variation ensures a realistic distribution of task complexity, from simpler stacking sequences to multi-step interdependent configurations. The goal is not to test scalability along a single axis (e.g., increasing object count), but rather to evaluate planner robustness across diverse scenarios with non-uniform decomposition depth and structural difficulty.

All experiments are conducted under consistent hardware and software conditions, identical to the development environment summarized in Table 5.1. This ensures that any observed performance differences are attributable solely to the planners' behavior and not to underlying system variability. For each problem instance, both planners generate structured output files in CSV format, from which three core evaluation metrics were extracted: *Planning Time*, defined as the time required to generate a valid plan (excluding DQL model inference overhead for DQL-SH); *Plan Length*, defined as the number of primitive actions in the final solution; and *Memory Usage*, measured as the peak memory consumption recorded during execution. To ensure fairness in comparison, the DQL-SH planner's reported planning time excludes the additional latency introduced by the external DQL model interface. These metrics form the basis for all subsequent statistical analysis and visualizations. To ensure reliable and uncontaminated measurements, each planner is executed independently on every problem instance, using a clean process for each run. This prevents residual memory or cached computations from affecting performance metrics. The resulting per-instance measurements are presented as a detailed plot in the next section, enabling visual comparison across the full benchmark set before moving on to aggregated statistics and formal significance testing.

6.2 Granular Results Across All Test Problem Instances

As a starting point for the evaluation, this section presents a comprehensive overview that jointly visualizes the planners' performance across all 100 benchmark test cases on the three core evaluation metrics: *Planning Time*, *Memory Usage*, and *Plan Length*. Figure 6.1 displays a unified line plot spanning the full set of test instances, from *blocksworld_test_700.log* through *blocksworld_test_799.log*, with each planner executed independently under identical hardware and software configurations. To improve readability without sacrificing resolution, only the numerical suffix of each filename (e.g., 700, 705, 710) is shown on the x-axis at five-instance intervals. This format enables clear visual comparison while preserving the detailed structure of the underlying performance data. This unified visualization serves as a foundation for the analysis by highlighting patterns, divergences, and fluctuations in performance, thereby preparing the ground for the more detailed statistical and qualitative evaluations that follow.

Planning Time. The top panel of the figure illustrates the time taken to compute a valid plan (in milliseconds) by each planner. The DQL-SH planner exhibits consistently faster planning across the majority of the test cases. Peaks in the SH curve, representing planning slowdowns due to inefficient task decomposition or increased search overhead, are often absent or significantly diminished in the DQL-SH curve. Particularly in cases such as *test_732*, *test_758*, and *test_786*, the reduction in planning time achieved by DQL-SH is both substantial and consistent. These gains reflect the model's ability to avoid costly decompositions by leveraging learned preferences for method selection.

Memory Usage. The middle panel reports the peak memory consumption (in megabytes) for each planner during the execution of a single planning episode. Across most instances, SH exhibits significantly higher memory usage, often reaching or exceeding 60 MB due to increased computational demands arising from deeper task hierarchies, expanded search spaces, and less efficient decision-making. In contrast, the DQL-SH planner maintains a flatter, lower profile around 27–30 MB across the board, highlighting the memory efficiency benefits of learning-based policies. The divergence becomes more evident in challenging instances, where task complexity and larger object sets naturally increase planning demands, making the benefits of guided method selection more apparent.

Plan Length. The bottom panel tracks the number of primitive actions in the resulting plan. While plan lengths between SH and DQL-SH often converge for simpler problems, DQL-SH frequently produces shorter plans as the task complexity increases. This suggests that the DQL-enhanced method selection is not only faster and more memory-efficient but also capable of guiding the planner toward more optimal decompositions. Notable examples of plan reduction can be seen in instances such as *test_741*, *test_768*, and *test_794*, where the DQL-SH agent consistently chooses action sequences that yield equivalent goal satisfaction in fewer steps.

Overall, this per-instance comparison offers strong empirical evidence in favor of the DQL-SH framework. Not only does it outperform the baseline SH planner across the majority of cases in planning time and memory usage, but it also demonstrates structural advantages in the compactness of the resulting plans. In a small number of instances, SH exhibits comparable or slightly better performance, particularly in terms of memory usage or plan length. These cases likely reflect situations where no significantly shorter decomposition was available, or where the model did not select the most optimal method. Nonetheless, these isolated outcomes do not detract from the broader trend: the DQL-SH planner consistently delivers superior performance and generalizes well across varying problem complexities. This detailed view affirms the effectiveness of learning-based decision-making and strengthens the foundation for the broader statistical results presented next.

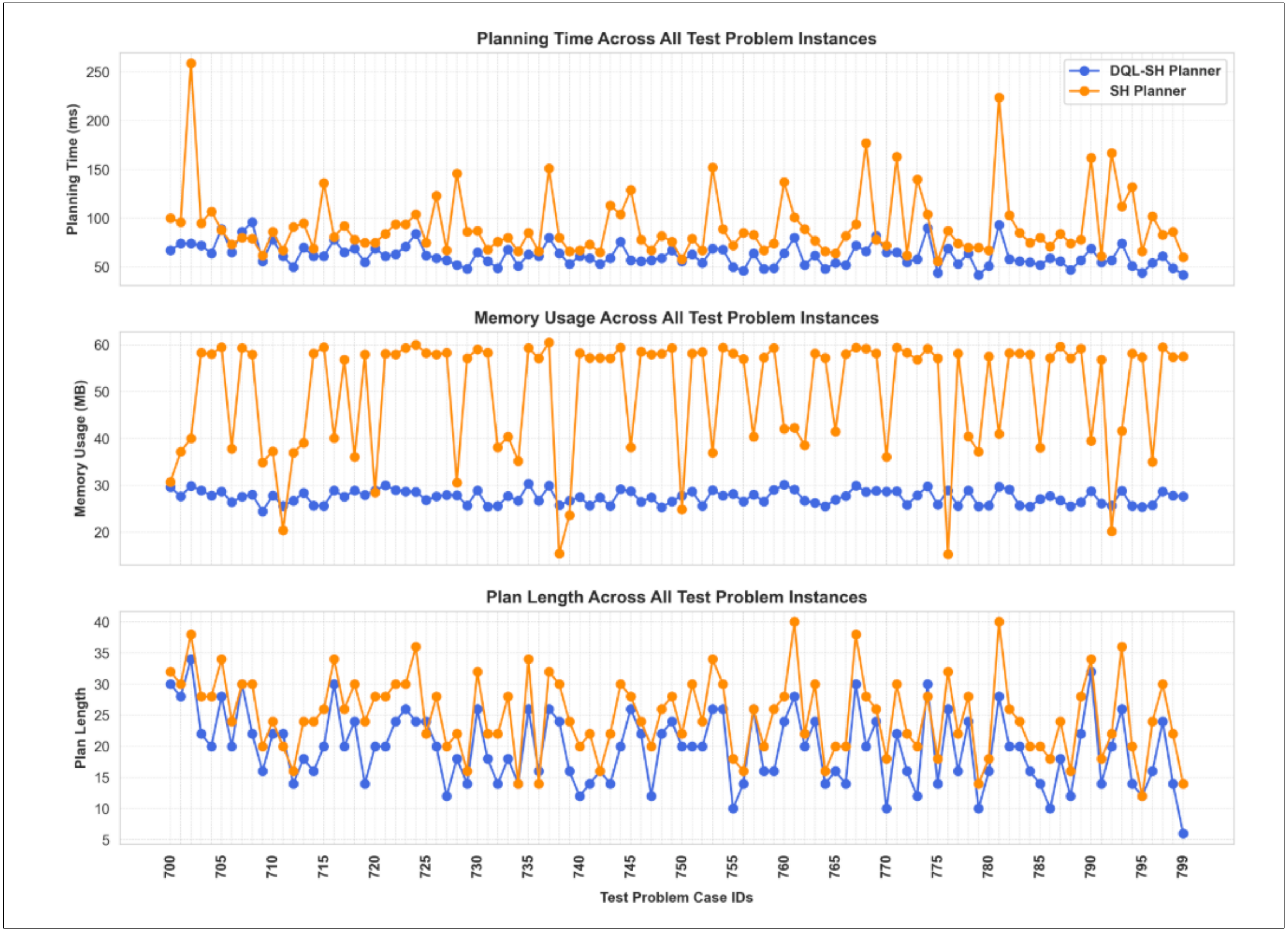


Figure 6.1: Planner Performance Trends Across 100 Test Cases. Line plots show per-instance comparisons between SH and DQL-SH across three core metrics: planning time, memory usage, and plan length.

6.3 Descriptive Statistics for Planner Performance

To quantitatively assess the comparative behavior of the baseline SH and the proposed DQL-SH planners, this section presents a descriptive summary of their performance across three core metrics: *planning time*, *memory usage*, and *plan length*. The analysis includes standard statistical descriptors, *mean*, *standard deviation (Std)*, *minimum (Min)*, and *maximum (Max)*, computed across 100 problem instances. These indicators collectively provide insight into not only average performance, but also variability, consistency, and robustness. The *mean* captures central tendency, while the *standard deviation* quantifies dispersion. The *min* and *max* values reflect the best- and worst-case outcomes, respectively. This statistical overview complements the earlier per-instance trends and lays the groundwork for the visual and inferential analyses presented in the following sections.

As shown in Table 6.1, DQL-SH consistently outperforms the baseline SH planner across all three metrics. In terms of *Planning Time*, DQL-SH achieves a 32.9% reduction in the mean (61.75 ms versus 91.97 ms), along with a 66.7% reduction in standard deviation (11.37 versus 34.12), highlighting both faster and more stable planning. Similarly, average *Memory Usage* drops by 44.8% (27.49 MB versus 49.79 MB), with an 88.0% reduction in variability (Std: 1.46 versus 12.21). For *Plan Length*, DQL-SH produces solutions that are 21.1% shorter on average (19.74 versus 25.02), while also slightly reducing variance (5.86 versus 6.3). Together, these statistics show that DQL-SH improves both computational efficiency and output stability, essential qualities for reliable and scalable planning systems.

Metric	**Mean**		**Std**		**Min**		**Max**	
	SH	DQL-SH	SH	DQL-SH	SH	DQL-SH	SH	DQL-SH
Planning Time (ms)	91.97	61.75	34.12	11.37	56	42	259	96
Memory Usage (MB)	49.79	27.49	12.21	1.46	15	24	60	30
Plan Length	25.02	19.74	6.3	5.86	12	6	40	34

Table 6.1: Descriptive statistics (mean, std, min, max) for SH and DQL-SH across all evaluation metrics.

To complement the numerical summaries, Figures 6.2 and 6.3 visualize the differences in mean and standard deviation across the three metrics. Each bar plot provides a side-by-side comparison of SH and DQL-SH, offering a more intuitive perspective on performance trends. The mean bar chart (Figure 6.2) highlights DQL-SH's consistent advantage, especially in planning time and memory usage. The standard deviation chart (Figure 6.3) further supports this pattern, showing that DQL-SH not only improves performance but also reduces variability, leading to more predictable outcomes across test cases.

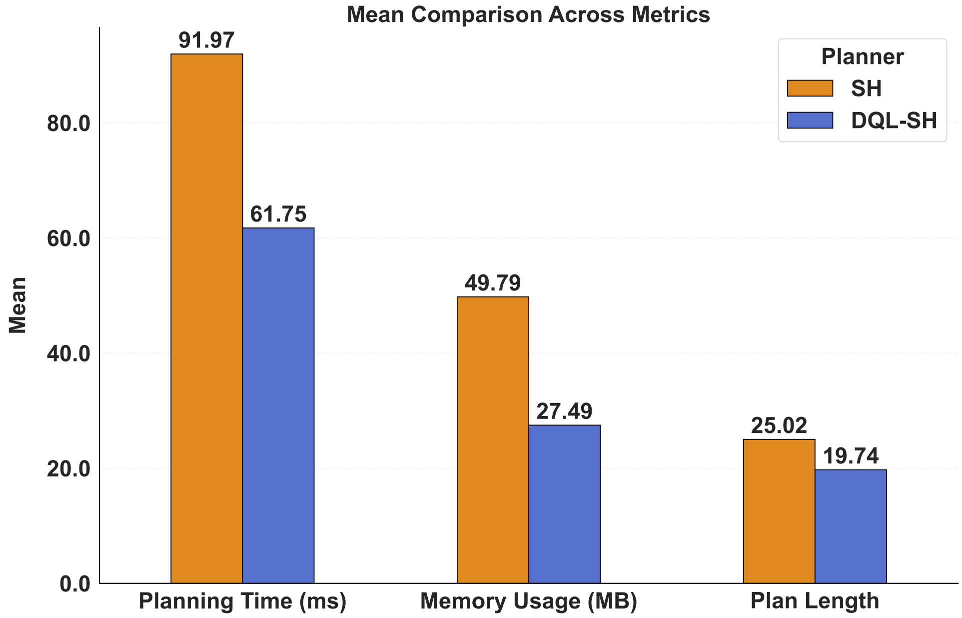


Figure 6.2: Bar chart comparing the mean values of SH and DQL-SH planners across three key metrics.

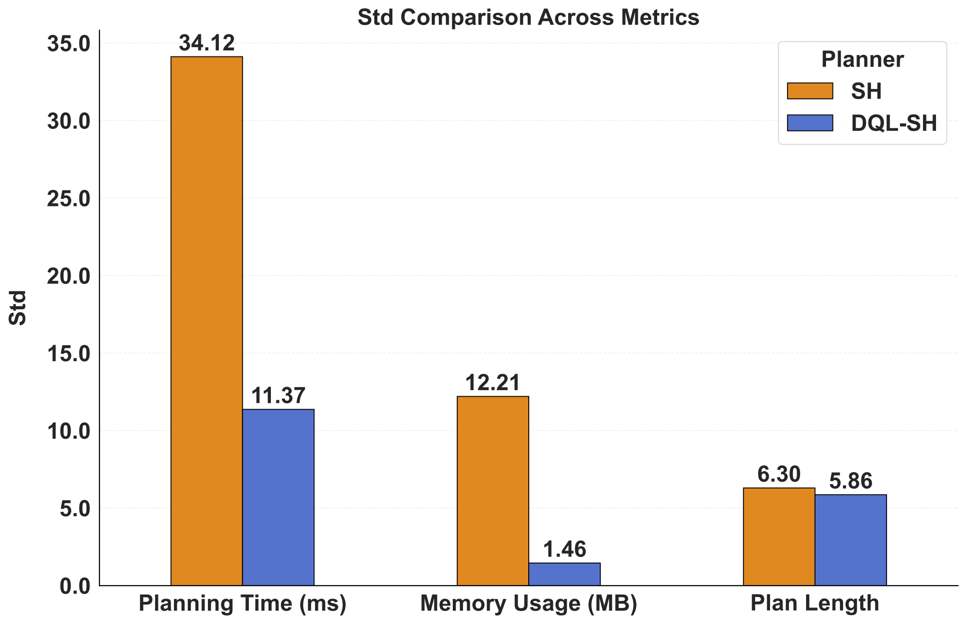


Figure 6.3: Bar chart comparing the standard deviation (Std) for SH and DQL-SH across three metrics.

6.4 Per-Metric Impact Analysis

In this section, we take a closer look at how learning-based method selection influences the planner's behavior across individual evaluation metrics. While the overall trends are already evident from the descriptive statistics and trend charts, this deeper investigation allows us to observe the specific impact of DQL integration along three critical dimensions: planning time, memory usage, and plan length. Each metric highlights a different aspect of planning performance, from temporal efficiency to resource consumption and plan compactness.

To support this analysis, we include two types of visualizations for each metric: a *horizontal boxplot*, which highlights the central tendency, spread, and outliers for both SH and DQL-SH; and a *kernel density estimate (KDE)*, which illustrates the overall distribution and concentration of values. These two visualizations clarify how the learning-enhanced planner compares to the baseline across both central tendencies and distributional properties. Taken together, they provide a detailed view of metric-wise behavior, enabling us to confirm broader trends while also uncovering finer-grained patterns across individual planning instances.

6.4.1 Planning Time

Planning time is a crucial metric in real-time and large-scale planning applications, as it directly reflects the efficiency of the underlying search process. Shorter and more consistent planning times typically indicate effective pruning of unpromising branches and faster convergence to a valid solution. As shown in Figure 6.4, subfigure *(a)* presents a *horizontal boxplot*, where DQL-SH exhibits a noticeably lower median planning time, narrower interquartile range, and fewer high-end outliers compared to the baseline SH planner. Subfigure *(b)* displays the corresponding *KDE*, which reveals that the distribution of DQL-SH is sharply peaked at a lower time range, indicating that most planning instances are solved more quickly. Together, these visualizations confirm that the learning-enhanced planner consistently selects effective decompositions earlier, thereby reducing search effort and overall execution time.

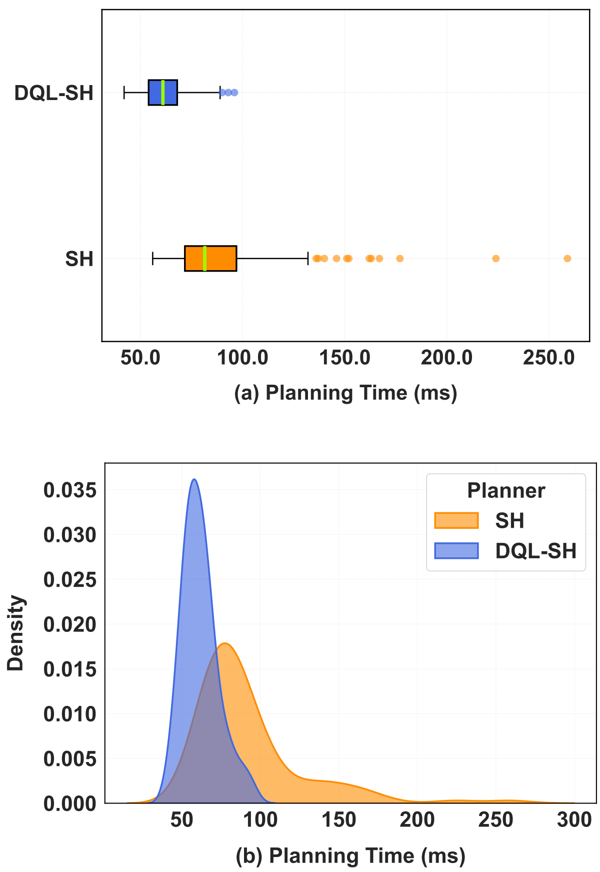


Figure 6.4: (a) Horizontal boxplot and (b) KDE plot comparing planning time for SH and DQL-SH.

6.4.2 Memory Usage

Memory efficiency is a key factor in the scalability of a planning system, particularly when dealing with complex or large-scale problem instances. Excessive memory consumption can slow down execution or lead to system failures due to resource exhaustion. A planner that minimizes memory overhead is better equipped to handle a wider range of domains and deeper decompositions. As shown in Figure 6.5, subfigure *(a)* presents a *horizontal boxplot* of memory usage, where DQL-SH demonstrates a clearly lower median and a more compact distribution compared to the baseline SH planner. The boxplot also reveals fewer extreme outliers, indicating greater consistency in resource consumption. Subfigure *(b)* displays the corresponding *KDE*, which shows a noticeable shift of DQL-SH's density curve toward the lower end of the memory scale. This concentration suggests that the learning-based method selector helps to reduce memory usage by guiding the planner away from unnecessarily complex or memory-intensive decomposition paths. These observations confirm the conclusion that DQL-SH not only improves planning efficiency but also enhances system stability by keeping memory demands under control.

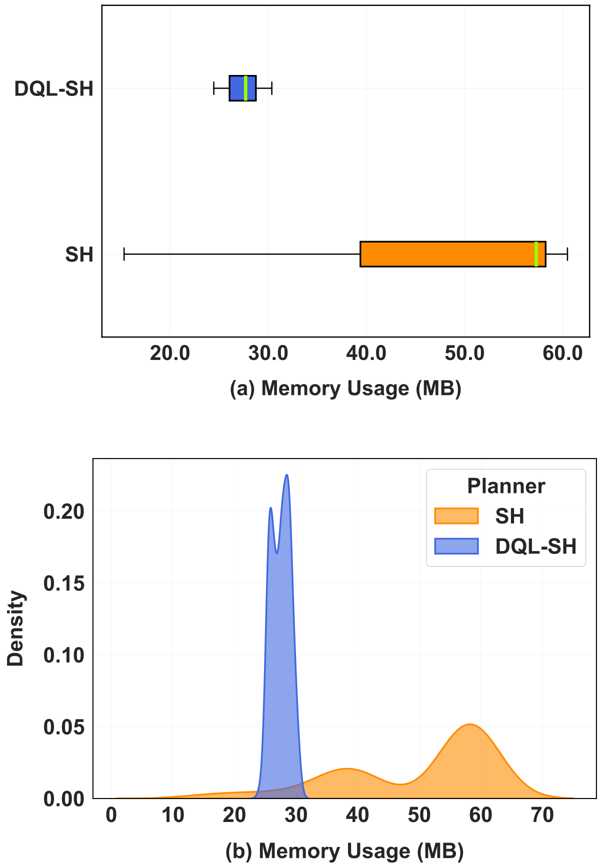


Figure 6.5: (a) Horizontal boxplot and (b) KDE plot comparing memory usage for SH and DQL-SH.

6.4.3 Plan Length

Plan length, defined as the total number of primitive actions in the final solution, serves as a proxy for both execution time and overall plan complexity. shorter plans generally reflect more direct or efficient decompositions, which are often desirable in practical applications where interpretability and execution cost matter. As shown in Figure 6.6, subfigure *(a)* presents a *horizontal boxplot* of plan length, where DQL-SH exhibits a slightly lower median and reduced spread relative to the baseline SH planner.

Moreover, the DQL-enhanced planner shows fewer high-end outliers, indicating that it is less prone to generating overly long solutions. Subfigure *(b)* displays the corresponding *KDE*, which reveals that the majority of DQL-SH plans are concentrated around a lower action count. This distribution suggests that the learning-based method selection strategy encourages more compact plans by avoiding unnecessary decompositional depth.

Overall, these findings complement the descriptive statistics by revealing how performance improvements unfold across individual instances. The visualizations confirm that DQL-SH consistently achieves lower medians, reduced variability, and more compact distributions across all metrics. These results provide a deeper understanding of planner behavior and support the empirical foundation for the significance testing that follows.

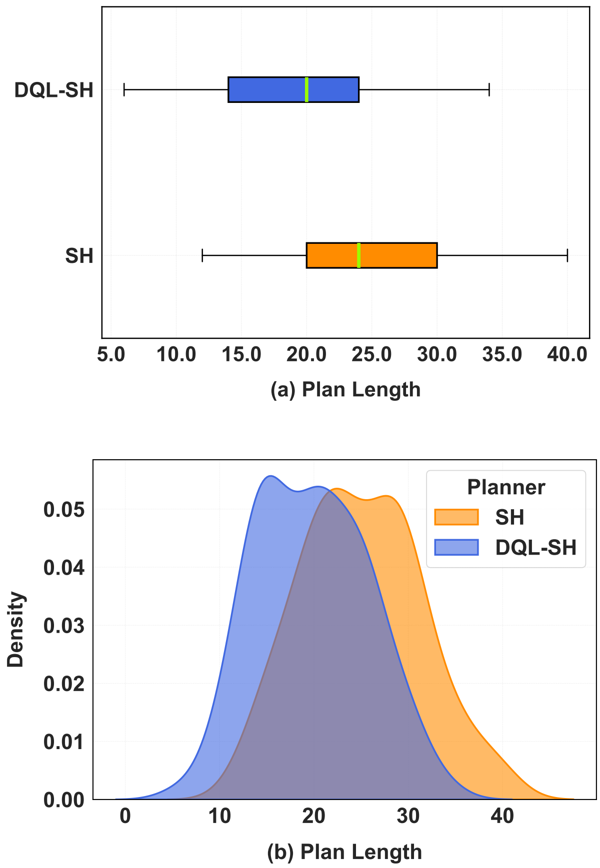


Figure 6.6: (a) Horizontal boxplot and (b) KDE plot comparing plan length for SH and DQL-SH.

6.5 Statistical Significance Testing

While descriptive and visual analyses indicate that DQL-SH outperforms the baseline SH planner, it is essential to determine whether these observed improvements reflect genuine performance gains or could have arisen by chance. Statistical testing offers a reliable way to judge whether the observed improvements are meaningful. In this section, we formalize the notion of paired performance differences between the two planners and define the associated null and alternative hypotheses. We then select appropriate statistical tests, both parametric and nonparametric, based on their assumptions and robustness. Finally, we report and interpret the resulting p-values to assess whether the differences across planning metrics are statistically significant.

6.5.1 Paired Differences and Hypothesis Testing

To provide a robust statistical foundation for our comparison, we consider the per-instance performance differences between the two planners. For each evaluation metric M and problem instance $i \in \{1, \ldots, N\}$, we define 6.1:

$$\Delta_i = M^{(i)}_{\mathbf{DQL}} - M^{(i)}_{\mathbf{SH}}, \tag{6.1}$$

where Δ_i represents the observed gain or loss in performance when replacing SH with DQL-SH. These paired differences, defined in Equation 6.1, reduce the two-sample comparison to a single-sample analysis of differences [CHW21]. This formulation allows us to construct a hypothesis test to determine whether the observed differences reflect a consistent trend or could be attributed to random variation. Specifically, we evaluate the null hypothesis that the expected difference is zero, against the alternative that a systematic effect exists. This hypothesis formulation, shown in Equations 6.1 and 6.2, provides the formal basis for the significance tests conducted in the following analysis [CHW21].

$$\text{Hypotheses:} \quad \begin{cases} H_0 : \mathbb{E}[\Delta_i] = 0 & \text{(no systematic difference)} \\ H_1 : \mathbb{E}[\Delta_i] \neq 0 & \text{(a systematic difference exists)} \end{cases} \tag{6.2}$$

To evaluate these hypotheses, we apply two complementary statistical tests, one parametric and one nonparametric. This dual approach ensures that our conclusions are robust across varying distributional assumptions of the paired differences $\{\Delta_i\}$. The first test is the *paired t-test*, a widely used parametric method that assumes the paired differences are approximately normally distributed. It evaluates whether the mean of $\{\Delta_i\}$ deviates significantly from zero. This test is particularly appropriate when the sample size is moderate to large and the data does not exhibit substantial skewness or heavy tails [CHW21]. To address potential deviations from normality, we also apply the *Wilcoxon signed-rank test*. This nonparametric test makes no assumptions about the underlying distribution and instead tests whether the median of $\{\Delta_i\}$ differs from zero. It is especially robust in the presence of outliers or skewed data, as it compares ranked differences rather than raw values [CHW21]. Both tests are conducted at a significance level of $\alpha = 0.05$. A p-value below this threshold indicates a statistically significant difference, while $p < 0.001$ is interpreted as very strong evidence against the null hypothesis. These thresholds help determine whether the improvements observed for DQL-SH are likely to reflect real performance gains rather than random variation.

The results of both statistical tests, applied across all evaluation metrics, are presented in Table 6.2. For each metric, planning time, memory usage, and plan length, both the paired t-test and the Wilcoxon signed-rank test yield p-values well below the conventional significance threshold of 0.05, confirming that the improvements introduced by DQL-SH are statistically significant. Notably, all p-values are on the order of 10^{-15} or smaller, offering overwhelming evidence in favor of the alternative hypothesis. This level of statistical confidence strongly suggests that the observed performance gains are not due to random variation. Moreover, the agreement between the parametric and nonparametric tests supports the robustness of the results. Together, these findings support the conclusion that the improvements in planning efficiency and plan compactness result from the integration of learning-based method selection, rather than stochastic effects.

Metric	**T-Test p-value**	**Sig (t)**	**Wilcoxon p-value**	**Sig (W)**
Planning Time	1.3352e-15	Yes	1.8254e-17	Yes
Memory Usage	3.3409e-33	Yes	3.7502e-17	Yes
Plan Length	3.3828e-30	Yes	6.0568e-17	Yes

Table 6.2: Significance test results for SH versus. DQL-SH across all metrics.

6.6 Planning Quality Assessment

In this section, we assess the quality of the plans generated by the DQL-SH planner in comparison to those produced by the baseline SH system. We focus on *plan length* as an indicative measure of planning quality, based on the premise that shorter plans are likely to contain more efficient action sequences and reflect more strategic decision-making during decomposition. While statistical significance testing in the previous section confirmed that the observed reductions in plan length are unlikely to be due to chance, our aim here is to qualitatively examine whether those shorter plans correspond to meaningful improvements in action selection and goal-directedness. We argue that the plans produced by the DQL-SH planner tend to follow more efficient paths through the task space, avoiding unnecessary or counterproductive steps. To support this claim, we now examine a representative Blocksworld case selected at random from the benchmark set, comparing plans generated by DQL-SH and SH side by side.

Case Study: `blocksworld_test_727.log`

As a qualitative example, we consider a Blocksworld planning problem involving the construction of two independent towers. The first tower is composed of block `b002` on `b003`, continuing down to `b006`, `b004`, and finally `b000`. The second tower stacks block `b007` on `b005`, which is itself placed on `b001`. This setup increases the planning complexity by introducing two disjoint substructures that must be assembled concurrently. The goal specification and the corresponding plans are shown in Figure 6.7.

The execution of the SH planner (bottom left) reflects inefficiencies in the way actions are selected and sequenced to achieve the goal. a primary inefficiency lies in the presence of redundant operations, particularly consecutive stack–unstack cycles applied to the same block in consecutive steps. For instance, the planner stacks `b004` on `b000` at step 2, only to unstack it again by step 3 and eventually place it down in step 4. Similar inefficiencies occur in steps 6–8 and 10–12. These inefficiencies reflect a lack of foresight and result in unnecessary backtracking, increased plan length, and reduced overall plan efficiency.

In contrast, the DQL-SH planner (bottom right) produces a significantly more coherent and streamlined plan. Rather than applying decomposition methods in a fixed or rule-based order, it relies on a trained Q-network to select the most promising method at each decision point based on the current planning state. This context-aware method selection enables the planner to avoid destructive sequences, such as stacking a block only to unstack it shortly afterward, which are common in the SH plan. As a result, each block is manipulated only when necessary and typically remains in its target position once placed. The plan eliminates redundant steps, and converges more directly toward the goal configuration, ultimately achieving a 40% reduction in plan length for this problem task.

This case study offers a clear qualitative demonstration of how learning-based method selection in DQL-SH results in more deliberate, context-aware decision-making that avoids redundant operations and maintains consistent progress toward the goal. The resulting plan reflects not only structural efficiency but also a deeper alignment between local decisions and global objectives, thereby confirming the quantitative trends observed in the broader evaluation.

Goal Set

1. (do_put_on b004 b000)
2. (do_put_on b006 b004)
3. (do_put_on b003 b006)
4. (do_put_on b002 b003)
5. (do_put_on b005 b001)
6. (do_put_on b007 b005)

SH Plan

1. unstack(b004,b002)
2. **stack(b004,b000)**
3. unstack(b004,b000)
4. put-down(b004)
5. unstack(b006,b001)
6. **stack(b006,b004)**
7. unstack(b006,b004)
8. put-down(b006)
9. pick-up(b003)
10. **stack(b003,b006)**
11. unstack(b003,b006)
12. put-down(b003)
13. pick-up(b002)
14. **stack(b002,b003)**
15. pick-up(b005)
16. **stack(b005,b001)**
17. unstack(b005,b001)
18. put-down(b005)
19. pick-up(b007)
20. **stack(b007,b005)**

DQL-SH Plan

1. unstack(b004,b002)
2. **stack(b004,b000)**
3. unstack(b006,b001)
4. **stack(b006,b004)**
5. pick-up(b003)
6. **stack(b003,b006)**
7. pick-up(b002)
8. **stack(b002,b003)**
9. pick-up(b005)
10. **stack(b005,b001)**
11. pick-up(b007)
12. **stack(b007,b005)**

Figure 6.7: Blocksworld Case Study (blocksworld_test_727.log): (Top) goal set. (Bottom left) SH plan and (Bottom right) optimized DQL-SH plan.

7 Conclusion and Future Work

This thesis presents a neuro-symbolic planning framework for enhancing method selection in state-based HTN planning through the integration of deep reinforcement learning. By embedding a trained DRL agent into the HTN planning system, the approach introduces a data-driven decision layer that complements traditional symbolic reasoning with learned behavioral patterns. This integration addresses a core limitation of classical HTN planners, their dependence on static, handcrafted heuristics, by enabling dynamic and context-sensitive method choices during decomposition.

A modular architecture was developed in which the SH planner interacts with the learning model via a RESTful FastAPI interface, allowing efficient integration without disrupting the planner's deterministic reasoning core. To support offline training, a structured data generation pipeline was implemented, transforming raw execution traces into high-quality training data. This pipeline incorporates a hierarchical representation of planning states and actions, ensuring compatibility with deep learning models while preserving semantic relevance.

Comprehensive experiments were conducted on a diverse set of Blocksworld planning problems, a classical benchmark domain, comparing the baseline SH planner with its learning-enhanced counterpart across multiple metrics, including planning time, memory usage, and plan length. The results demonstrate that the learning-based approach leads to meaningful improvements in efficiency and solution quality, particularly in complex scenarios with high method branching. Although the learned model occasionally performs on par with the baseline in simpler cases, the overall enhancement trend is clear and statistically significant, affirming the value of integrating reinforcement learning into HTN planning.

Looking ahead, this work opens several directions for future research. One promising avenue is to extend the framework to better handle varying degrees of method alternativity, including weak alternatives that are not always mutually exclusive or equally applicable. Capturing such nuanced decision contexts may require richer state representations and could further improve method selection in structurally diverse domains. Additionally, exploring alternative reinforcement learning architectures, such as dueling DQNs or actor-critic models, may enhance the system's ability to prioritize between competing decomposition strategies. Finally, applying the approach to broader domain sets would help assess its generality and adaptability across planning scenarios.

In summary, this thesis demonstrates that DRL can effectively augment classical HTN planning by enabling more adaptive, robust, and scalable method selection. The resulting hybrid system combines the strengths of symbolic structure with data-driven learning, contributing to the advancement of intelligent planning in dynamic and complex environments.

Bibliography

[AG23] E. Alnazer, I. Georgievski. “Understanding real-world ai planning domains: A conceptual framework”. In: *Symposium and Summer School on Service-Oriented Computing*. Springer. 2023, pp. 3–23 (cit. on p. 32).

[AG24] J. Alammar, M. Grootendorst. *Hands-on large language models: language understanding and generation*. Ö’Reilly Media, Inc.", 2024 (cit. on p. 48).

[AGA22] E. Alnazer, I. Georgievski, M. Aiello. “On bringing HTN domains closer to reality-the case of satellite and rover domains”. In: *International Conference on Automated Planning Systems (ICAPS) Workshop on Scheduling and Planning Applications (SPARK)*. 2022, p. 13 (cit. on p. 32).

[AGPA22] E. Alnazer, I. Georgievski, N. Prakash, M. Aiello. “A role for HTN planning in increasing trust in autonomous driving”. In: *2022 IEEE International Smart Cities Conference (ISC2)*. IEEE. 2022, pp. 1–7 (cit. on p. 13).

[AR18] B. Andrew, S. Richard S. *Reinforcement learning: an introduction*. The MIT Press, 2018 (cit. on pp. 22, 30).

[BAH19] P. Bercher, R. Alford, D. Höller. “A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations.” In: *IJCAI*. 2019, pp. 6267–6275 (cit. on p. 18).

[BBHB17] P. Bercher, G. Behnke, D. Höller, S. Biundo. “An Admissible HTN Planning Heuristic.” In: *IJCAI*. 2017, pp. 480–488 (cit. on pp. 14, 26).

[CHW21] D. Childs, B. Hindle, P. Warren. *APS 240: Data analysis and statistics with R*. 2021 (cit. on p. 63).

[CWY+18] K. Cheng, L. Wu, X. Yu, C. Yin, R. Kang. “Improving hierarchical task network planning performance by the use of domain-independent heuristic search”. In: *Knowledge-Based Systems* 142 (2018), pp. 117–126 (cit. on pp. 14, 25).

[Datnd] Databricks. *Medallion Architecture*. https://www.databricks.com/glossary/medallion-architecture. Accessed: 2025-05-24. n.d. (Cit. on p. 36).

[DDD+20] H. Dong, H. Dong, Z. Ding, S. Zhang, T. Chang. *Deep reinforcement learning*. Springer, 2020 (cit. on pp. 21, 22).

[EGW05] D. Ernst, P. Geurts, L. Wehenkel. “Tree-based batch mode reinforcement learning”. In: *Journal of Machine Learning Research* 6 (2005) (cit. on pp. 23, 30).

[FL03] M. Fox, D. Long. “PDDL2. 1: An extension to PDDL for expressing temporal planning domains”. In: *Journal of artificial intelligence research* 20 (2003), pp. 61–124 (cit. on p. 20).

[FWXY20] J. Fan, Z. Wang, Y. Xie, Z. Yang. “A theoretical analysis of deep Q-learning”. In: *Learning for dynamics and control*. PMLR. 2020, pp. 486–489 (cit. on p. 22).

[GA15] I. Georgievski, M. Aiello. “HTN planning: Overview, comparison, and beyond”. In: *Artificial Intelligence* 222 (2015), pp. 124–156 (cit. on pp. 13, 18).

[GB21] I. Georgievski, U. Breitenbücher. “A vision for composing, integrating, and deploying AI planning functionalities”. In: *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2021, pp. 166–171 (cit. on p. 29).

[Geo13] I. Georgievski. “HPDL: Hierarchical planning definition language”. In: *Uni. of Groningen, JBI Preprint* (2013), pp. 12–3 (cit. on p. 20).

[Geo15] I. Georgievski. “Coordinating services embedded everywhere via hierarchical planning”. In: (2015) (cit. on pp. 18, 21).

[Geo22a] I. Georgievski. “Office activity recognition using HTN planning”. In: *2022 16th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. IEEE. 2022, pp. 70–77 (cit. on p. 13).

[Geo22b] I. Georgievski. “Towards engineering ai planning functionalities as services”. In: *International Conference on Service-Oriented Computing*. Springer. 2022, pp. 225–236 (cit. on p. 29).

[Geo23] I. Georgievski. “Software Development Life Cycle for Engineering AI Planning Systems.” In: *ICSOFT*. 2023, pp. 751–760 (cit. on p. 29).

[Geo25] I. Georgievski. “Occupant Activity Recognition in IoT-Enabled Buildings: A Temporal HTN Planning Approach”. In: *International Conference on Agents and Artificial Intelligence*. 2025, pp. 29–32 (cit. on p. 13).

[GNLA17] I. Georgievski, F. Nizamic, A. Lazovik, M. Aiello. “Cloud ready applications composed via HTN planning”. In: *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2017, pp. 81–89 (cit. on p. 13).

[GNT04] M. Ghallab, D. Nau, P. Traverso. *Automated Planning: Theory and Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004. ISBN: 9780080490519. URL: https://books.google.de/books?id=uYnpze57MSgC (cit. on pp. 13, 17, 18).

[GPAA24] I. Georgievski, A. V. Palghadmal, E. Alnazer, M. Aiello. “SH: Service-oriented system for HTN planning in real-world domains”. In: *SoftwareX* 27 (2024), p. 101779 (cit. on pp. 20, 21, 30, 40).

[GVAM20] M. Gregurić, M. Vujić, C. Alexopoulos, M. Miletić. “Application of Deep Reinforcement Learning in Traffic Signal Control: An Overview and Impact of Open Traffic Data”. In: *Applied Sciences* 10.11 (2020). ISSN: 2076-3417. DOI: 10.3390/app10114011. URL: https://www.mdpi.com/2076-3417/10/11/4011 (cit. on p. 49).

[HBBB18] D. Höller, P. Bercher, G. Behnke, S. Biundo. “A generic method to guide HTN progression search with classical heuristics”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 28. 2018, pp. 114–122 (cit. on p. 25).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. “The PANDA framework for hierarchical planning”. In: *KI-Künstliche Intelligenz* 35.3 (2021), pp. 391–396 (cit. on p. 18).

[HKM10] C. Hogg, U. Kuter, H. Munoz-Avila. “Learning methods to generate good plans: Integrating htn learning and reinforcement learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 24. 1. 2010, pp. 1530–1535 (cit. on pp. 14, 27, 40).

[HMK16] C. Hogg, H. Muñoz-Avila, U. Kuter. “Learning hierarchical task models from input traces”. In: *Computational Intelligence* 32.1 (2016), pp. 3–48 (cit. on p. 26).

[JK19] J. M. Johnson, T. M. Khoshgoftaar. “Survey on deep learning with class imbalance”. In: *Journal of big data* 6.1 (2019), pp. 1–54 (cit. on p. 50).

[MBR+25] A. Micheli, A. Bit-Monnot, G. Röger, E. Scala, A. Valentini, L. Framba, A. Rovetta, A. Trapasso, L. Bonassi, A. E. Gerevini, et al. “Unified Planning: Modeling, manipulating and solving AI planning problems in Python”. In: *SoftwareX* 29 (2025), p. 102012 (cit. on p. 13).

[MCNF17] M. Mukadam, A. Cosgun, A. Nakhaei, K. Fujimura. “Tactical decision making for lane changing with deep reinforcement learning”. In: (2017) (cit. on p. 27).

[MGJC18] A. Menif, C. Guettier, E. Jacopin, T. Cazenave. "Applying Anytime Heuristic Search to Cost-Optimal HTN Planning". In: *Computer Games: 6th Workshop, CGW 2017, Held in Conjunction with the 26th International Conference on Artificial Intelligence, IJCAI 2017, Melbourne, VIC, Australia, August, 20, 2017, Revised Selected Papers 6*. Springer. 2018, pp. 151–171 (cit. on pp. 14, 25).

[MKS+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533 (cit. on pp. 22, 23, 30, 49).

[NAI+03] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, F. Yaman. "SHOP2: An HTN planning system". In: *Journal of artificial intelligence research* 20 (2003), pp. 379–404 (cit. on p. 13).

[NCLM99] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila. "SHOP: Simple hierarchical ordered planner". In: *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. 1999, pp. 968–973 (cit. on p. 13).

[NFP22] C. Núñez-Molina, J. Fernández-Olivares, R. Pérez. "Learning to select goals in Automated Planning with Deep-Q Learning". In: *Expert Systems with Applications* 202 (2022), p. 117265 (cit. on p. 26).

[OAC+04] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger. "An overview of the Scala programming language". In: (2004) (cit. on p. 29).

[PMC23] R. F. Prudencio, M. R. Maximo, E. L. Colombini. "A survey on offline reinforcement learning: Taxonomy, review, and open problems". In: *IEEE Transactions on Neural Networks and Learning Systems* (2023) (cit. on p. 21).

[PyT24] PyTorch Contributors. *torch.utils.data.WeightedRandomSampler — PyTorch 2.4 Documentation*. https://docs.pytorch.org/docs/stable/data.html#torch.utils.data.WeightedRandomSampler. Accessed: 2025-04-05. 2024 (cit. on p. 50).

[RCX09] S. Reiff-Marganiec, K. Chen, J. Xu. "Markov-htn planning approach to enhance flexibility of automatic web services composition". In: (2009) (cit. on pp. 14, 26).

[Soh19] S. Sohrabi. "AI Planning for Enterprise: Putting Theory Into Practice." In: *IJCAI*. 2019, pp. 6408–6410 (cit. on p. 17).

[Str25] P. Strengholt. *Building Medallion Architectures: Designing with Delta Lake and Spark*. O'Reilly Media, 2025. ISBN: 9781098178802. URL: https://books.google.de/books?id=bFtSEQAAQBAJ (cit. on p. 36).

[SZC+21] T. Shao, H. Zhang, K. Cheng, K. Zhang, L. Bie. "The hierarchical task network planning method based on Monte Carlo Tree Search". In: *Knowledge-Based Systems* 225 (2021), p. 107067 (cit. on pp. 14, 26).

[SZXD17] Y. Shen, N. Zhao, M. Xia, X. Du. "A deep q-learning network for ship stowage planning problem". In: *Polish Maritime Research* 24.s3 (2017), pp. 102–109 (cit. on p. 27).

[VFP20] I. Vellido, J. Fdez-Olivares, R. Pérez. "A knowledge based process for the generation of htn domains from vgdl video game descriptions". In: *ICAPS2020 Workshop on Knowledge Engineering for Planning and Scheduling*. 2020, pp. 11–20 (cit. on p. 20).

[WD92] C. J. Watkins, P. Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292 (cit. on pp. 22, 30).

[WKK08] N. Waisbrot, U. Kuter, T. Könik. "Combining Heuristic Search with Hierarchical Task-Network Planning: A Preliminary Report." In: *FLAIRS*. 2008, pp. 577–578 (cit. on pp. 14, 25).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature