ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Laurea Magistrale in Ingegneria e Scienze Informatiche

Exploiting GenAI for Plan Generation in BDI Agents

Tesi di laurea in:
INTELLIGENT SYSTEMS ENGINEERING

Relatore
Prof. Giovanni Ciatto

Correlatore
Prof. Gianluca Aguzzi

Candidato
Riccardo Battistini

I Sessione di Laurea
Anno Accademico 2024–2025

Abstract

Extending BDI agents with the ability to autonomously generate plans has long been a goal in cognitive agent engineering, aiming to improve their adaptability in dynamic environments. Recent advances in GenAI offer promising new opportunities in this area by leveraging the natural language understanding, means-end reasoning, and abstraction capabilities of LLMs. This thesis explores the integration of GenAI-driven plan generation into AgentSpeak(L) agents, examining how knowledge can be effectively transferred between the LLM and the BDI agent to support dynamic, runtime plan creation. To this end, a novel framework is proposed that extends the AgentSpeak(L) reasoning cycle with generative capabilities, enabling agents to synthesize plans on-the-fly. The design and implementation of this generative process are discussed, along with the architectural modifications required. The framework is implemented using the JaKtA BDI interpreter, and the quality of the generated plans is systematically evaluated across different types of LLMs.

Contents

Abstract **iii**

1 Introduction **3**

2 Background and Related Work **5**
2.1 Planning in AgentSpeak(L) agents . . . 5
2.2 JaKtA: Jason-like Kotlin Agents . . . 9
2.3 Plan Generation in BDI Agents . . . 9
2.4 Reasoning and Planning in LLMs . . . 11
2.5 GenAI Agents . . . 12
2.6 Integration of BDI agents with LLMs . . . 14

3 Design **15**
3.1 The structure of a PGP . . . 15
3.2 Triggering the generative process . . . 17
3.2.1 On-demand PGP . . . 17
3.2.2 Reactive PGP . . . 18
3.2.3 Proactive PGP . . . 19
3.3 Concurrency and PGP . . . 20
3.4 Bridging BDI Agents and LLM Knowledge . . . 21
3.4.1 From BDI Agents to LLM and Back . . . 21
3.4.2 What to Encode . . . 21
3.4.3 How to Encode . . . 23
3.5 Writing Generative Agent Specifications . . . 25

4 Implementation **27**
4.1 Integration with the BDI Engine . . . 27
4.1.1 The PGP contract . . . 28
4.1.2 The Generation Manager . . . 31
4.2 Logging System . . . 36
4.3 Generative Process Pipeline . . . 39

4.3.1 Context filters . . . 39
4.3.2 Formatters . . . 40
4.3.3 Parsers . . . 40
4.3.4 Request Handlers . . . 41
4.3.5 Plan Generators . . . 42
4.4 Generative Agent Specification . . . 42
4.4.1 Implementing Custom Filters . . . 43
4.4.2 Defining Prompt Builders . . . 44
4.4.3 Declaring the Generation Strategy . . . 44
4.4.4 Writing the Documentation . . . 46
4.4.5 Specifying on-demand Generation Goals . . . 47
4.4.6 Defining Custom Log Events . . . 48

5 Evaluation **51**
5.1 The explorer robot application . . . 51
5.1.1 Agent . . . 51
5.1.2 Environment . . . 53
5.2 Experimental Methodology . . . 53
5.2.1 Experimental Setup . . . 54
5.2.2 Language Models . . . 55
5.2.3 Experiments' Parameters . . . 56
5.2.4 Evaluation Metrics . . . 56
5.3 Experimental Results . . . 58
5.3.1 Example PGP Prompt . . . 58
5.3.2 Example PGP Responses . . . 61
5.3.3 Detailed Metrics' Analysis . . . 61

6 Conclusion **75**
6.1 Future Work . . . 76
6.1.1 Plan repair and refinement mechanisms . . . 77
6.1.2 Structured Output Formats . . . 77
6.1.3 Improved Software Modularity . . . 78
6.1.4 Model Context Protocol Integration . . . 79
6.1.5 Finetuning Small Language Models . . . 79
6.1.6 Artifacts as Tools . . . 80

Bibliography **81**

List of Figures

4.1 Simplified BDI control cycle that uses a reactive PGP to dynamically generate new plans at runtime. 28
4.2 The interfaces that define the contract for plan generation exposed by the belief-desire-intention (BDI) engine and the interfaces and classes that implement this contract for a LM-based PGP. 29
4.3 Components of the `GenerationManager`, which orchestrates the plan generation procedure (PGP). 31
4.4 Components of the `GeneratePlanStrategy`, which invokes the generation strategy provided by the user and updates the result to the agent that requested the generation. 35
4.5 `LogEvent` interface hierarchy. 37
4.6 `AgentEvent` interface hierarchy. 37
4.7 `EnvironmentChange` interface hierarchy. 37
4.8 `ExecutionFeedback` interface hierarchy considering only goals' results. 38
4.9 `ExecutionFeedback` interface hierarchy considering only PGPs' results. 38
4.10 Set of interfaces that handle the parser's results. 41
4.11 Set of classes that compose a LLM-based plan generation strategy. . 42

5.1 Environment modelling for the explorer robot domain. The agent is in discrete bidimensional space, where it can move in eight admissible directions, namely: the cardinal ones, with north pointing up. Direction `here` denotes the current position of the agent. . . . 52
5.2 The average **PRAS** score for each model was calculated by sampling up to five PGP results from the set of successful PGP invocations. 70
5.3 Bar plots showing **PC**, **CC**, **PBC**, **GC**, **RR**, **NGC**, **NBC** and **GSA** scores for each model, with a facet for each prompt type tested. 71

5.4 Bar plots showing **PC**, **CC**, **PBC**, **GC**, **RR**, **NGC**, **NBC** and **GSA** scores for each model, with a facet for each value of temperature tested. 72
5.5 **BSA** and **TSR** scores for each model, with a facet for each prompt type tested. 73
5.6 **BSA** and **TSR** scores for each model, with a facet for each value of temperature tested. 73

List of Listings

3.1 One possible implementation of a reactive PGP 18
3.2 Fragment of the format restricting instructions that define the hybrid YAML and Prolog syntax that the LLM is expected to use to format its responses in the implemented PGP prototype. 24
3.3 An example of how a description enriches an admissible goal with an explanation of its meaning, to the benefit of both the human programmer and the large language model (LLM)-based PGP. . . . 26
4.1 Implementation of the action formatter that converts `Actions` into human-readable string representations for LLM prompts. 40
4.2 Default context filter implementation that removes meta-plans and failure-handling plans to streamline the LLM context for plan generation. 43
4.3 Example of a user prompt template, which is populated at runtime according to the state of the agent and the content of static files. . 45
4.4 Example of a generation strategy declared at the MAS level. 46
4.5 Example of a generation strategy declared at the agent level. 46
4.6 Example of a generation strategy declared at the plan level. 47
4.7 Example of a JaKtA agent extended with natural-language descriptions 47
4.8 Example of a JaKtA agent that will try to generate a plan to print the numbers from zero to ten using the provided generation strategy. 48
4.9 Implementation of the `move` action, which provides a more specific feedback than the default `GenericActionSuccess` when the action completes successfully. 49
5.1 The `ExplorerRobot` agent as implemented in JaKtA. 52
5.2 The set of percepts generated by the grid world environment. . . . 53
5.3 Specification of the grid world environment in JaKtA. 54
5.4 Set of baseline plans for the ExplorerRobot agent 55
5.5 Prompt used for the LLM-as-judge evaluation. 57
5.6 The system prompt used by the robot explorer application. 59
5.7 The user prompt used by the robot explorer application. 60

5.8 Example response given by Claude Sonnet 4 (beautiful-volhard). This is one of the few successful PGP invocations which use a prompt with no hints or remarks. 61
5.9 Example response given by Deepseek V3 Chat (stoic-rhodes). . . . 62
5.10 Example response given by GPT 4.1 (stupefied-lewin). 63
5.11 Example response by Gemini 2.5 Flash (frosty-kare). This is the response that got the highest score by the LLM judge, which coincidentally was Gemini 2.5 Flash itself, and was one of the few answers the model provided that were successful. 64

Acknowledgements

I would like to thank my supervisor Giovanni Ciatto and co-supervisor Gianluca Aguzzi, for their guidance and support throughout this thesis. I am also thankful to the doctoral candidates Martina Baiardi and Samuele Burattini for their suggestions and feedback, which helped shape the research direction. Additional thanks go to Professor Alessandro Ricci for his insightful input during our group meetings. I am grateful for the opportunity to collaborate with them on a paper, based on the thesis idea presented here, that has been accepted for publication at the 28th European Conference on Artificial Intelligence (ECAI-2025).

Last but not least, I want to thank my family for their support, patience, and encouragement throughout this master's program.

Chapter 1

Introduction

Long before the current surge of interest in generative AI (GenAI), the BDI model has worked as a paradigm for rational agents, leading to the development of several BDI agent-oriented programming architectures [1], as well as agent programming languages based on them—most notably, AgentSpeak(L) [2]. BDI languages promote the design of rational agents able in principle to reason about complex and dynamically evolving environments, and proactively act upon them.

Nevertheless, traditional BDI programming frameworks typically lack mechanisms allowing agents to autonomously acquire or build new plans at runtime [3]. The procedural behavior of AgentSpeak(L) agents is then limited to scenarios where the agent behavior can be suitably defined at design time (pre-programmed): an effective option in terms of computational efficiency, which however constrains agent flexibility, responsiveness, and overall *autonomy* when facing unpredictable environments. Extending BDI agents with planning capabilities has thus been extensively explored [4].

Prior approaches typically attempt to bridge first-principles planning techniques with BDI architectures [3], requiring a detailed model of the action space. However, those models, usually expressed in terms of preconditions and effects, are typically costly and often unavailable for highly-dynamic environments.

Recent advances in GenAI methodologies promise to overcome such limitations by offering more dynamic and flexible solutions. GenAI technologies such as LLMs provide unprecedented opportunities for empowering cognitive agents with intelligent features involving the generation of cognitive abstractions (such as beliefs, goals, plans) as well as the ability to interact in natural language.

The exploration conducted in this thesis is grounded on the assumptions that: *(i)* LLMs exhibit planning capabilities [5]; *(ii)* LLMs easily handle natural language, and have absorbed *common sense* knowledge that can be leveraged in the planning process [6]; *(iii)* LLMs systems exhibit abstraction and *imagination* capabilities—e.g. [7], which may lead to more flexible plans than those generated by classical planning techniques. Furthermore, given that BDI developers typically employ mnemonic and semantically-relevant names resembling natural language to represent cognitive abstractions, this work explores whether LLMs can generate BDI plans by leveraging the natural-language semantics inherent in goals, actions, and plans, potentially reducing reliance on explicitly-defined preconditions and effects.

This work focuses on the problem of augmenting BDI agents with autonomous plan generation capabilities by leveraging GenAI. The investigation seeks to answer these four research questions: **(RQ1)** *what* information would LLMs require to generate BDI plans? **(RQ2)** *how* should knowledge be transferred between LLMs and BDI agents? **(RQ3)** *how* does automatic plan generation impact BDI agents operation and specification? **(RQ4)** *can* LLMs generate *reusable* BDI plans? To address them, this thesis analyzes the integration requirements for incorporating generative AI into BDI architectures, proposes a conceptual framework that extends the traditional BDI model to support automated plan generation, and presents preliminary experimental results obtained through a prototype implementation in JaKtA [8].

This work is organized as follows. Chapter 2 provides background on BDI agents and on their reasoning and planning capabilities, reviews related work on planning in BDI systems, and introduces GenAI agents. Chapter 3 presents the design of the generative process for runtime plan generation. Chapter 4 offers a detailed examination of how GenAI-based plan generation procedures are integrated into AgentSpeak(L) agents. Chapter 5 discusses the experimental results, while chapter 6 concludes the work and outlines directions for future research.

Chapter 2

Background and Related Work

This chapter presents the theoretical and technical foundations relevant to this thesis. Section 2.1 introduces the BDI model and its application in programming AgentSpeak(L) agents. Section 2.3 discusses plan generation within BDI architectures. Section 2.4 provides a short review of the reasoning and planning capabilities demonstrated by LLMs. Recent developments in agent architectures based on generative AI are outlined in section 2.5. Lastly, section 2.6 surveys existing efforts to integrate BDI models with LLMs.

2.1 Planning in AgentSpeak(L) agents

The BDI model, inspired by human cognitive processes [9], and grounded upon a formal framework [10] for modelling *rational agents*' decision-making, has been adopted for agent programming languages—e.g. [11]—as well as for simulating intelligent agents [12] or human behavior [13]. BDI systems represent a wide yet specific class of multi-agent systems (MASs), where rational agents feature *cognitive* abstractions such as beliefs, goals, and intentions.

Specifically, a BDI agent [1]: *(i)* maintains *beliefs* about itself and its environment, updated by perception, reasoning, or communication with other agents; *(ii)* is guided by the *desires* (or goals) it aims to achieve, which may evolve over time or give rise to new desires during their pursuit; *(iii)* commits to multiple concurrent *intentions* as a means to fulfil its desires; *(iv)* executes *plans*—as sequences of *actions* it knows how to perform—in order to achieve its goals.

Several architectures have been proposed to implement BDI agents in software, including AgentSpeak(L) [1], procedural reasoning system (PRS) [14], and distributed Multi-Agent Reasoning System (dMARS) [15]. For an account of the many architectures proposed in the literature refer to the survey by Meneguzzi and de Silva [16]. In the following, the focus is on the widely-adopted AgentSpeak(L)—so that the terms BDI agents and AgentSpeak(L) agents may be used interchangeably from now on.

AgentSpeak(L) agents are animated by events, to which they respond by selecting *plans* that guide their execution. Events may correspond to the addition or removal of beliefs—which, in turn, may be the result of perception or communication with other agents—or to the occurrence of internal events, such as the commitment of the agent to a new goal to pursue. Unlike purely reactive systems, where external stimuli are directly mapped to actions, BDI agents reason about events in the context of their current state and choose *plans* accordingly, via a continuous *deliberation* activity. The execution of plans can, in turn, generate new internal events, allowing the agent to progress through its reasoning cycle and operate autonomously.

AgentSpeak(L) agents perform so-called *procedural reasoning* [14]: they are equipped by human developers with procedural knowledge in terms of a *plan library*, a collection of plans agents can select for execution at run time. Agents' deliberation process is then aimed at selecting the most adequate plan for any event that may occur, so that plans are, ultimately, a way to encode specific actions to perform in response to specific events, failing if none has been provided.

More precisely, AgentSpeak(L) plans are referred to as *plan rules*, denoted as: `E: C ← A`$_1$`; …; A`$_n$`.` The rule describes how the agent should react to some *triggering event* `E`, stating for instance what to do when a new percept arises, or how to achieve a given goal. `C` is the set of conditions under which the rule is applicable (a.k.a. the rule's *context*), and `A`$_1$`; …; A`$_n$ is the *body* of the rule, corresponding to the actual plan, i.e., the actions to perform to handle the event.

Plan rules hence differ significantly from first-principles planning, where the plan is a sequence of actions to be performed to reach a desired world state—i.e., *declarative* goals [17]. In fact, although it is possible to write plans rules and goals in a declarative style to mimic classic planning [18, 19], it is often the case that a goal expressed as the triggering event of a BDI plan rule is simply a *mnemonic name* of an event the agent may be willing to react to—i.e., *procedural* goals [17]. Additionally, other than encoding the procedural knowledge to achieve goals (`!Goal`), AgentSpeak(L) plan rules can be written to react to the addition or removal of beliefs (`+/-Bel`), as well as performing *epistemic actions*—e.g., a *test* to retrieve information from the current belief base (`?Test`). *(i)* The addition (resp. removal) of a belief `+Bel` (resp. `-Bel`) representing some novel (resp. old) information which the agent memorized (resp. forgot) due to reasoning, perception, or communication; or *(ii)* the commitment to a new achievement (resp. test) *goal* `+!G` (resp. `+?G`), representing something that the agent intends to do (resp. know). Practically speaking, while beliefs are logic (most commonly, Prolog) facts,—i.e. the agents' belief base is considered as a logic theory—such as `temperature(25, celsius)`, goals are logic terms named after the activities they represent, such as `!go(home)` or `?weather(raining)`. Plans events may be non-ground, in which case the plan would react to any event matching its head via logic unification—e.g., the plan `+temperature(X, celsius): tolerance(T) & (X > T) ← !go(away)` would match any update concerning the temperature. As shown in the last example, plans' contexts are just logic expressions possibly involving the variables in the plan's head, or any other belief from the agent's belief base. The many activities in a plan's body can either be *(i)* subgoals, i.e., expressions such as `+!G` or `+?G`, representing new goals to be achieved (or tested) by the agent; *(ii)* actions, i.e., logic terms denoting operations to be performed, such as `move(up)`; or *(iii)* special actions for the update of believes (e.g. `+Bel` or `-Bel`). Actions can, and most commonly will, provoke side effects when executed, i.e. change either the agent's internal status or affect the environment.

Similarly to goals, actions are basically *mnemonic names* of functionalities causing effects. Unlike other artificial intelligence (AI) subfields, though, pre-/post-conditions for an action are not explicitly represented in AgentSpeak(L) as the actions are usually invoking low-level software/hardware facilities that bring side effects about. Typically, in BDI agents, actions may even fail due to unpredictable environment dynamics making pre-/post-conditions hard to define.

Lastly, a notable feature of AgentSpeak(L) plan rules is that the *body* of a plan may contain subgoals—i.e., statements triggering the selection and execution of other plans. Through this mechanism, AgentSpeak(L) agents can pursue complex goals by breaking them down into smaller subgoals for which the specific plans are selected at runtime, resulting in a non-linear execution. Yet, the underlying assumption is that all plans are known in advance, and the agent can select them based on the current context. Then, it may happen that a plan is not available for a given event and context, in which case the agent may fail to achieve the goal. To make that possible, AgentSpeak(L) assumes that low-level software/hardware facilities are attached to the agent, and that executing the action will invoke them accordingly. So, in a sense, actions are invocations to foreign language interfaces (FLI).

When encountering a sub-goal during plan execution, a new internal event is generated inside the agent, which may trigger the execution of another plan, and so on recursively. Stacks of goals, subgoals, and partially executed plans are called *intentions*. In a given moment, each agent may have multiple intention, each one keeping track of some course of action the agent is currently carrying on.

In this framework, goals are just *mnemonic names* of events the agent may be willing to react to, and, similarly, actions are just *mnemonic names* for functionalities which provoke side effects. However, differently from other subfields of AI, the actual semantics of goals and actions (namely, what properties of the world should be reached for a goal to hold, or what pre-/post-conditions is an action subject to) is not explicitly represented in AgentSpeak(L). So, it is up to human developers to write *meaningful* plans, goals, and actions, similarly to what programmers do with ordinary programming languages.

By supporting multiple concurrent intentions, BDI agents can manage complex scenarios where multiple goals must be pursued in parallel, and where multiple courses of actions must be interleaved to pursue them all. Yet, the underlying assumption is that all plans are known in advance, and the agent can select them based on the current context. It may happen that a plan is not available for a given event and context, in which case the agent may fail to act, and the whole intention may remain unfulfilled.

2.2 JaKtA: Jason-like Kotlin Agents

JaKtA [8] is an internal Domain-Specific Language (DSL) designed for implementing BDI agents directly in Kotlin. Unlike traditional BDI frameworks which rely on external DSLs, JaKtA is built using Kotlin's native language features as an internal DSL. This approach leverages Kotlin's expressive syntax and type system to provide a more integrated and type-safe environment for Agent-oriented Programming (AoP).

JaKtA includes a BDI interpreter that is compliant with AgentSpeak(L), enabling developers to define agents, beliefs, goals, plans, and actions as first-class Kotlin constructs. It draws inspiration from Jason [20], a widely used Java-based framework for BDI agent programming based on an extended version of AgentSpeak(L). While Jason provides a solid foundation for developing multiagent systems, its external DSL approach requires developers to work with a separate language syntax and toolchain. JaKtA addresses this by embedding AoP capabilities directly into Kotlin's syntax, allowing developers to benefit from Kotlin's modern language features such as null safety, coroutines, and functional programming constructs. This internal DSL approach enables more seamless integration between agent logic and conventional programming constructs, reducing the cognitive overhead of context switching between different syntaxes. Additionally, Kotlin's interoperability with Java ensures that JaKtA maintains access to the extensive JVM ecosystem while providing a more contemporary development experience, by taking advantage of IDE features, debugging tools, and build systems. The framework's design also facilitates the combination of AoP with other programming paradigms within the same codebase, leading to more sophisticated and maintainable agent-based systems. Due to these design advantages and its alignment with modern development practices, JaKtA has been chosen for the development of the PGP prototype.

2.3 Plan Generation in BDI Agents

This section highlights the main implications of integrating plan generation techniques into BDI agents, which serves as a foundation for this work as well.

The main limitation of procedural reasoning in BDI agents is that the plan library is statically defined by the agent programmer, which has to anticipate all the possible situations the agent may encounter. Although this is beneficial for the agent's predictability and controllability, it limits the agent's flexibility and adaptability to unforeseen situations. For this reason, the BDI community has long been interested in the problem of *automatic plan generation*, incorporating techniques from AI planning into the BDI architecture. For an in-depth overview of the many approaches proposed in the literature so far, refer to the survey by Meneguzzi and de Silva [4].

Section 2.1 surfaced the main differences between BDI plans and classical (a.k.a. first-principles) planning (e.g. STRIPS [21]) which can be essentially be summarized as: *(i)* BDI plans and goals are procedural, while in classical planning they are declarative [17]; *(ii)* BDI actions have no clearly stated pre- and post--conditions, while classical planning actions are defined by them *(iii)* BDI plans can include subgoals for a more flexible execution flow, while classical planning is usually assembling a linear sequence of actions to move from one world state to another. Hierarchical Task Network (HTN) planning [22], based on the idea of task decomposition, allows generating more abstract plans that can mimic the invocation of subgoals and thus addressing item *(iii)*. Nevertheless, tasks must be defined in advance, have pre- and post-conditions and potentially ordering constraints. Interestingly, as emerging from [4], classic planning techniques have mostly been integrated to generate new plans, while HTN has been exploited to perform look-ahead in the plan library and understand if a suitable chain of plans is available.

Regardless of the specific planning technique, integrating plan generation in BDI agents requires deciding *when* to trigger the generation process. The most straightforward approach is to trigger the generation process when the agent is unable to find a suitable plan in its library. Another common approach is to trigger the generation through an agent action that can be invoked intentionally by the agent programmer as part of a plan [23]. The plans generated can then either be used to extend the original plan library, or be executed immediately (or both). In the first case, plans should ideally be reused in the future [24], although this may lead to a bloated plan library and conflicts with the original plans that need to be handled [25].

2.4 Reasoning and Planning in LLMs

LLMs demonstrate remarkable proficiency in performing complex mathematical and logical operations, suggesting sophisticated reasoning abilities [26]. The introduction of Chain-of-Thought (CoT) prompting has further amplified these capabilities by encouraging models to break down complex problems into intermediate reasoning steps, leading to substantial improvements in performance on challenging tasks [27, 28]. However, while LLMs exhibit impressive problem-solving behavior, questions persist on the degree of robustness and generality of such abilities, both for what concerns LLMs and large reasoning models (LRMs) [29, 30, 31].

Recent work by Shojaee et al. [32] highlights a critical limitation of LLM reasoning by systematically analyzing performance under controlled complexity in symbolic problem domains such as Tower of Hanoi and Blocks World [33]. They identify a three-phase behavior in LRMs: dominance over non-thinking models on moderately complex tasks, regression on simpler tasks due to "overthinking", and total failure as problem complexity escalates. Despite models having computational budget, they reduce reasoning effort under higher complexity, revealing a scalability ceiling in inference-time reasoning.

Complementing these findings, Kambhampati et al. argue that LLMs fundamentally lack planning and self-verification capacities due to their architectural limitations and reliance on "approximate retrieval" of similar reasoning patterns in their training data [34, 35]. Through empirical evidence from PlanBench and planning domain benchmarks, they show that even the most advanced models fail to generate executable plans or reliably critique their outputs. In experiments with the Blocks World domain, LLMs were tasked with generating valid action sequences to reach specific goal configurations. Despite using various prompting techniques (zero-shot, one-shot, Chain-of-Thought), success rates remained modest; for instance, GPT-4 achieved only 34.3% correctness in one-shot mode. The performance degraded significantly when the task was presented in a more syntactically obfuscated variant called Mystery Blocks World, where action and object names were replaced with arbitrary tokens. In this case, even top-performing models dropped to near-zero accuracy (e.g. GPT-4 scored 4.3% in one-shot and 0.16% in zero-shot). These results support the hypothesis that LLMs often rely on surface-level pattern matching and approximate retrieval rather than performing true reasoning or plan synthesis. The drastic performance collapse in the obfuscated version indicates a brittle dependence on lexical familiarity and highlights the lack of robust abstraction capabilities essential for planning. As an alternative approach, they propose the *LLM-Modulo* framework, which treats LLMs as approximate knowledge sources embedded in a generate-test-critique loop with external verifiers. This hybrid neuro-symbolic approach allows LLMs to play constructive roles in reasoning pipelines.

These works suggest that while LLMs can mimic certain aspects of reasoning behavior—particularly when supported by carefully designed prompts—they do not possess robust, generalizable reasoning or planning abilities. Their effectiveness appears to depend significantly on context, complexity, and integration with structured external systems capable of ensuring correctness and coherence.

Given the ongoing debate surrounding the nature and reliability of LLM reasoning and planning capabilities, this thesis adopts a pragmatic stance: it builds on established best practices in prompt engineering and assumes that the current planning and reasoning capabilities of LLMs are adequate for generating plans within the context of BDI agent architectures. Nonetheless, chapter 6 outlines potential avenues for extending this framework in future work, particularly through the introduction of feedback loops—drawing inspiration from the LLM-Modulo paradigm—to enhance the robustness and reliability of plan generation.

2.5 GenAI Agents

Recent advances in GenAI have sparked significant interest in developing agents that incorporate LLMs. These systems leverage LLMs as cognitive engines, utilizing their natural language processing, reasoning, planning, and knowledge recall abilities to create agents capable of complex, autonomous behavior [36, 37, 38]. The remarkable ability of LLMs to perform effectively across diverse domains without task-specific training has made them ideal foundations for generative agents and autonomous systems [39]. In these agentic architectures, LLMs serve as the central "brain" that processes information, formulates plans, and coordinates actions. This integration enables the emergence of sophisticated behaviors that extend far beyond simple text generation, encompassing goal-directed planning, multistep reasoning, and adaptive problem-solving. The field has witnessed the development of various paradigms, including generative agents, agentic AI, and autonomous agents, that explore how LLMs reasoning can be harnessed for intelligent behavior [40]. Several approaches have emerged within this broad category.

One common approach focuses on creating "general-purpose agents" [40] capable of executing actions, managing a memory, and using software tools with minimal human intervention. Works following this approach, e.g. Auto-GPT [41], demonstrate that LLMs can autonomously decompose goals into actionable steps.

Another approach aims to situate LLMs in physical environments. For instance, Voyager [42] represents a significant step towards lifelong learning agents by autonomously exploring the complex environment of Minecraft, acquiring new skills through interaction and feedback, and storing them in a skill library for future use. Similarly, SayCan [43] and PaLM-E [44] integrate LLMs with robotic control, using them to interpret high-level natural language instructions and generate action plans for execution.

Rather than proposing entirely new agent architectures, some works focus on enhancing the reasoning capabilities of existing ones. The Reasoning and Acting (ReAct) framework [45] proposes a method for LLMs to synergize reasoning (generating thought traces) and acting (performing actions, e.g. tool use) in an interleaved manner, enabling agents to dynamically plan, execute, and adjust based on observations. Similarly, Reflexion [46] introduces agents that can reflect on past failures, maintain dynamic memory, and use self-reflection to improve their decision-making capabilities over time. While these approaches focus primarily on enhancing reasoning capabilities, the generative agent architecture underlying "Generative Agents" [36] provides a blueprint for single agents exhibiting believable, human-like, means-end reasoning capabilities. These agents leverage memory structures (capturing past observations and deliberations) and LLMs-driven planning to simulate daily routines, form relationships, and generate emergent social dynamics based on their individual experiences and environment.

2.6 Integration of BDI agents with LLMs

Several approaches explore combining BDI principles with LLMs. The work by [47] proposes NatBDI, a BDI agent architecture designed for natural language environments. It leverages LLMs, specifically for natural language inference (NLI), within the BDI reasoning cycle to determine the applicability of plans based on natural language beliefs and context descriptions, rather than generating the plans themselves. Another direction, presented by [48], uses the BDI structure itself to create structured prompts ("BDIPrompting") for LLMs. This guides the LLM to generate more proactive and explainable task plans by framing the planning problem in terms of beliefs, desires, and intentions directly within the prompt given to the LLM.

The approach presented in this thesis diverges by focusing on integrating generative plan generation into the well-established AgentSpeak(L)-based architecture. While much contemporary research prioritizes creating cognitive agents that directly leverage GenAI—where agents *are* generative systems—this work adopts a complementary perspective: strategically embedding generative capabilities within the BDI framework rather than replacing it. This design preserves the controllability, predictability, and explainability of agent behavior while leveraging generative capabilities [49].

Recent work has proposed enabling traditional agents to interrogate LLMs on-demand for plan adaptation in Web environments [50]. However, this thesis focuses on generating reusable plans from existing agent knowledge rather than relying on it to choose which actions to perform among those that are dynamically discovered at runtime.

Chapter 3

Design

This chapter proposes a conceptual framework that extends the BDI model to support plan generation through LLMs. It defines *how* the generative process works (§3.1), along with *when* it should be triggered (§3.2), and how the overall AgentSpeak(L) architecture should be adapted to accommodate generative capabilities. Concurrency considerations are examined in section 3.3. Section 3.4 addresses knowledge exchange between the agent and the LLM, and section 3.5 presents guidelines for writing generative agent specifications.

3.1 The structure of a PGP

This section explores how BDI agents can leverage LLMs to *dynamically* synthesize actionable plans tailored to their specific goals. This capability is enabled by endowing each agent with a *plan generation procedure (PGP)*, which the agent can invoke to generate new plans and incorporate them into its plan library.

Conceptually speaking, such a PGP involves *(i)* encoding the agent's current cognitive state and operational context—specifically: available actions, existing sub-plans, own beliefs, and the active goals the agent seeks to accomplish –into a structured prompt comprehensible to the LLM, and *(ii)* parsing the LLM output to extract the generated plans. The implementation of this PGP must ensure that the generated plans are not only syntactically valid, but also coherent and contextually relevant, effective and possibly efficient in achieving the goal(s) they have been generated for.

The PGP functionality works as follows: it accepts a goal `G` as input (be it an achievement `!g` or a test `?G` goal), and it returns a non-empty set of plans $\mathcal{P}$, $\equiv p_1, \ldots, p_m$, such that at least one plan $p \in \mathcal{P}$ handles the event `+G`. The plans in $\mathcal{P}$ are subject to the following constraints: **(C1)** each generated plan $p \in \mathcal{P}$ must handle different contexts (i.e., there should never be two plans $p, p' \in \mathcal{P}$ such that $p \neq p'$ yet both have the triggering event `E` and context `C`); **(C2)** the generated plans' contexts and bodies *should* refer[1] to other goals or beliefs that are already known to the agent, if these help pursuing the goal `G`; **(C3)** yet they may also reference novel, unknown goals or beliefs; **(C4)** the generated plans' bodies *must not* refer to actions[2] which are unknown to the agent.

The rationale behind these constraints is straightforward: the PGP should handle the goal it was triggered for, and it should reuse existing knowledge *whenever possible*. In particular, if the generated plans need to affect the environment, they should do so by leveraging the actions the agent is already aware of—which were supposedly provided by the agent programmer(s) in advance. This implies that the PGP should keep into account which and how much information is available to the agent, and use it in the generation process. Yet, the PGP should also be open to the generation of new knowledge in the form of beliefs, and goals. This may open up to the possibility, for the agent, to reuse the PGP multiple times,recursively generating hierarchies of interrelated plans.

[1] In the context of AgentSpeak(L), plan contexts may *refer* to beliefs, for instance to check whether a certain condition holds. Similarly, plan bodies may *refer* to (sub-)goals, for instance, because the achievement of the (sub-)goal is a precondition for the accomplishment of the plan. In this work a belief is considered as already known to the agent, if it is present in the agent's belief base at run time. Analogously, a goal is considered as already known to the agent, if the agent's plan library has a plan that handles the goal.

[2] Plans bodies may also refer to actions. Similarly to subgoals, these are activities to be performed by the agent to accomplish the plan. Yet, differently than subgoals, actions are "executed" and when that happens, they affect the environment via actuators. It is assumed that a low-level implementation is available to the agent, which directs the actuators.

3.2 Triggering the generative process

Taking inspiration from prior literature on plan generation in BDI agents, three main approaches can be identified to trigger a PGP, namely: the on demand, reactive and proactive ones. Each approach comes with different assumptions and implications, concerning the role of the agents' programmer, the shape of the generated plans, and the autonomy of the agent.

3.2.1 On-demand PGP

The on-demand PGP is modelled as an *action* that the agent can invoke whenever it needs to generate a plan. Technically speaking, the agent is assumed to have a built-in action `generate_plan(E)` in its plan library, which it can invoke whenever it needs to generate a plan for event `E`. As an ordinary AgentSpeak(L) action, `generate_plan` may appear in the body of any plan, and be exploited by programmers to build smarter agents. The action would just require the agent to provide some actual event `E` as input, and its effect would be the production of a plan of the form `E ← C : …`, where `C` is the context of the plan, to be added to the invoking agent's plan library. Any failure in the PGP execution would be reported as a failure of the action, to be handled by the agent as it would do for any other action. The underlying assumption here is that the agent's programmer is in charge of deciding when then agent should trigger the PGP. In fact, by writing ordinary AgentSpeak(L) plans, programmers may decide to employ the `generate_plan` action to govern what exact plan should be generated, and under which conditions. Another consequence of this approach is that, by looking at an agent's plan library, one may easily identify which and how many goals are going to require plan generation. This may be useful for debugging purposes, or to understand the agent's behavior in advance.

It is worth noticing that, as a consequence of constraint **(C2)**, the PGP may generate plans that use the `generate_plan` action in their body. So, generated plans may themselves trigger the PGP to generate new plans, when executed. This powerful feature allows the agent to hierarchically generate plans, possibly decomposing complex plans into simpler ones, and so on recursively—provided that the underlying LLM is smart enough to understand when it is wise to postpone the generation of a (sub-)plan, and when it is better to generate it immediately.

3.2.2 Reactive PGP

The reactive PGP is implicitly triggered by the agent's control loop whenever an event `E` occurs and the agent has no relevant plan for it. So, an unknown event would trigger the PGP instead of making the current intention fail. If the PGP fails, the current intention fails—in the same way as when the agent has no plan available; if the PGP succeeds, the agent would keep on executing its intention as if the plan was already available from start.

Provided that agents are equipped with the aforementioned `generate_plan` action, one may consider implementing the reactive approach as shown in listing 3.1:

```
p_genex ≡ -G : missing_plan_for(G) <- generate(G); G.
```

Listing 3.1: One possible implementation of a reactive PGP

This approach borrows Jason's extended AgentSpeak(L) syntax [11] for failure handling plans to express a plan (namely, p_{genex}) reacting to the failure event provoked by the absence of a plan for goal `G`, to which the agent should react by generating a plan for it, and finally pursue goal `G` one more time. The underlying assumption here is that agents are equipped with the plan p_{genex}, as well as with the special belief `missing_plan_for(G)`, whether plans are missing in the agent's plan library for goal `G`.

The reactive approach improves the on-demand approach by making the generation procedure automatic, and therefore implicit. Neither the agent's programmer nor the LLM writing the plans have to worry about explicitly triggering the PGP: they just have to mention goals to pursue the plans, and the agent will take care of triggering the PGP whenever needed. In fact, another benefit here is the minimality of PGP executions: aside from explicit invocations, the PGP is triggered only when strictly necessary.

Finally, the reactive approach as well supports the generation of hierarchies of plans. In fact, as prescribed by constraint **(C3)**, the PGP may generate plans that refer to novel, unknown goals. When met at run-time, the agent will trigger the PGP again to generate a plan for the new goal, and so on recursively. The main difference here is that the LLM is not required to be aware of the recursive nature of the planning activity it is involved into. So, it can focus on the problem suggesting the best subgoals and actions to be performed to pursue a given goal.

3.2.3 Proactive PGP

In the proactive approach, the PGP is triggered by an ad-hoc, background intention, which is assumed to be built-in in each agent. The intention would be responsible for continuously monitoring the agent's internal state, looking for goals or events for which the agent has no plan, and triggering the PGP for each of them. In case that no plans are missing, the intention would simply suspend itself or do nothing, waiting for further plans or goals being added to the agent—as these may contain new goals or events which require plan generation.

Conceptually speaking, such an intention is never meant to terminate. Rather, it should carry on the following operations in a loop: *(i)* select one event `E` for which no plan is available in the agent's plan library; *(ii)* trigger the PGP to generate a plan for `E` on a new intention —so to allow multiple, concurrent PGPs for different events— *(iii)* wait for the addition of plans or goals in the agent's internal state; *(iv)* repeat. Borrowing again from Jason syntax, the proactive approach can be expressed as follows:

```
g_init   ≡ !generate_plans.

p_scan ≡ !generate_plans <-
      for (event(E) & missing_plan_for(E)) {
          !!pgp(E)
      }; !generate_plans.

p_gen   ≡ !pgp(E) <- generate_plan(E).
```

There, goal g_{init} is one built-in initial goal of the agent, whose only purpose is to start the generation intention along with the agent. Plan p_{scan} is a scanning plan: it selects known events from the agent's internal state—via the special belief `event(E)`, plus the aforementioned `missing_plan_for` one—and triggers plan p_{gen}

on a new intention for each of them. Plan p_{gen}, in turn, triggers the PGP to generate a plan for the event `E`, without executing it thereafter. Any failure in plan p_{gen} would not interrupt plan p_{scan}, as it is running on a different intention.

Similarly to the reactive approach, the proactive one allows for the generation of hierarchical plans, without requiring any explicit action by either the agent's programmer or the LLM. Yet, by anticipating the PGP execution, it may also mitigate the risk of the agent being stuck waiting for the PGP to terminate, while also being in *urgent* need of a plan to know how to act. The drawback however is that race conditions may occur, for instance between an intention needing some missing plan, and the intention in charge of generating it. So, it may happen that one intention fails out of a missing plan, because the intention in charge of generating it has not been executed yet. This is a problem that may be solved by the agent's programmer, at the expense of additional coding efforts.

3.3 Concurrency and PGP

Regardless of how/when the PGP is triggered, any implementation approach should take into account that the PGP is inherently *blocking* for the agent. In fact, at the current state of technology, LLM are commonly queried via Web services and they may take up to many seconds to complete their response or even fail to respond at all, e.g. due to network issues. Even by assuming that the BDI agent and the LLM are running on the same machine, the LLM may require entire seconds to generate a plan. Such large time scales may be unacceptable for BDI agents, especially if waiting for the PGP to terminate implies blocking the agent's control loop. This would hinder the agents' ability to react to events in a timely manner, and therefore its autonomy.

To mitigate this problem, implementers should consider *suspending* only the intention that triggered the PGP, rather than the whole control loop of the agent. As multiple intentions may be concurrently carried on by an AgentSpeak(L) agent, this would allow it to "keep thinking about how to react to event `E`, while doing something else".

3.4 Bridging BDI Agents and LLM Knowledge

This section explores how to effectively engineer the prompts for LLM to let it generate plans, and how to write agent specifications in AgentSpeak(L), to enrich them with additional knowledge that can be exploited by the generative process.

3.4.1 From BDI Agents to LLM and Back

When triggered for a goal `G`, the PGP should: *(i)* convert the agent's internal state, as well as `G`, into a textual prompt, which is then used to query some LLM of choice, then *(ii)* parse the LLM response to extract the generated plans. The specific LLM chosen does not impact the design of the PGP itself, yet it may affect the quality of the generated plans. This is further discussed in chapter 5.

So, *prompt generation* (structured information into free text) and *parsing* (free text back into structured information) are the two main non-trivial, *complementary* activities of PGP. They may be implemented in several ways, mostly differing in terms of *what to encode*, and *how to encode* it, in the LLM prompt and in the response.

3.4.2 What to Encode

This section describes the PGP backwards, from its end back to its start, to better understand the rationale behind the proposed encoding.

The Response

The PGP should produce a set of BDI plans by querying a LLM, which in turn would produce a textual response. The response should therefore represent a (possibly empty) collection of plans, where each plan should involve: *(i)* a triggering event, *(ii)* a context—i.e. a (possibly empty) collection of conditions to test –, and *(iii)* a body—i.e. a (possibly empty) collection of subgoals and actions.

The Prompt

To support the generation of the aforementioned plans, the PGP should provide all—and possibly only—relevant information to the LLM via some *automatically*-generated prompt, following a typical zero-shot prompting approach [51], therefore encoding some instruction and information about the current agent's state in natural language and in structural form. This directly addresses item **(RQ1)** (what information LLMs require).

The minimal set of relevant information might consist of: **(I1)** the intended meaning of the goal `G` for which the plans are needed along with the explicit request to generate plans for it, **(I2)** the goals, actions, and beliefs which are already known to the agent, **(I3)** the current plans, goals, and beliefs of the agent; as well as instructions on **(I4)** what is the intended outcome of the plan generation process, **(I5)** the AgentSpeak(L) syntax and its intended meaning, **(I6)** how to impersonate a BDI agent willing to generate a plan (i.e., role-playing prompting [52]), **(I7)** what a BDI agent is in the first place, by providing the general notions of beliefs, desires, and intentions, and **(I8)** the syntax the LLM should use to encode its responses.

Whereas the LLM needs item **(I1)** to know what goal the agent is willing to pursue, item **(I2)** is required to know which goals, actions, or beliefs (akin to tool specifications in ReAct [45]) the LLM could use to generate the plans.

For the LLM to account for the context currently perceived by the agent when generating the plans, item **(I3)** is needed. Item **(I4)** informs the LLM about the purpose and the constraints of the plan generation process, as defined in section 3.1. This is where, for instance, the possibility of formulating new goals and beliefs should be mentioned—along with the impossibility of creating new actions.

Item **(I5)** lets the LLM comply with the AgentSpeak(L) syntax, when both reading the prompt and producing its response, and makes it aware of how cognitive abstractions are modelled in AgentSpeak(L). This is where, for instance, the LLM is informed about the three parts of a plan (event, context, and body), and the components of a plan body (subgoals and actions). Along with item **(I7)**, the core idea here is to provide the LLM with enough background knowledge to operate in the BDI domain, regardless of whether BDI literature has been included in the training set of the LLM or not.

Item **(I6)** provides the LLM with meta-level suggestions about how to devise plans in general. There, recommendations such as "decompose the task hierarchically" or "re-use the known goals and actions whenever useful" could be included.

Finally, item **(I8)** defines the instructions used to nudge the LLM towards a response that is amenable to parsing.

It is worth noticing that, while item **(I1)**, item **(I2)**, and item **(I3)** are specific to the agent's current state, item **(I4)**, item **(I5)**, item **(I6)**, item **(I7)**, and item **(I8)** are more general, and could be factorized across different runs of the PGP.

3.4.3 How to Encode

In principle flexibility in syntax is allowed for both the prompt and the response, provided that *(i)* the syntax used for the prompt is easily manipulable by the LLM, *(ii)* the syntax used for the response is easily parsable by some parser not using LLMs.

When selecting the encoding format for LLM interactions, two main dimensions need to be taken into account. The first dimension concerns input structure complexity, spanning from highly structured programming languages through markup languages like YAML to weakly structured natural language. The second dimension addresses the strategies to obtain a *structured output* [53], with three primary approaches identified by [54]: *(i) constrained generation* using context-free grammars (e.g., JSON mode) or regular expressions [55], *(ii) format-restricting instructions* that guide LLMs toward standardized schemas like JSON, XML, or YAML, and *(iii)* natural language-to-format conversion, where models first generate natural language responses before transforming them into target formats.

These choices involve several trade-offs that impact the PGP performance. The first one is between output structure compliance and generation performance [54]: while highly structured formats ensure consistent parsing, they may constrain the model's capacity to express itself. Another is the complexity versus accessibility trade-off: sophisticated encoding schemes can capture rich semantics but often require specialized knowledge, whereas simpler formats are more broadly usable but may lack expressive power. Format familiarity also plays a significant role. Widely used schemas such as JSON and YAML, which frequently appear in training corpora, tend to yield more reliable outputs than niche formats like AgentSpeak(L), which have limited representation in LLM training data [56, 57]. Finally, there's a latency versus accuracy trade-off, especially in multistep approaches like natural language-to-format conversion, which can enhance output quality at the expense of increased processing time and computational overhead.

This work aims at maximizing the exploitation of natural language, as done in related works that employ LLM as a planner [40], to achieve *(i)*, by enriching the AgentSpeak(L) syntax with natural language descriptions. For requirement *(ii)*, format restricting instructions are employed to nudge the LLM towards structured output by providing a detailed description of the expected response format as part of the prompt. Listing 3.2 shows an example of this approach, used in the PGP prototype, to instruct the LLM to format its responses with a hybrid YAML and Prolog syntax.

````
## Operation Types
Operations must be prefixed with keywords:
- `execute`: primitive actions that directly interact with environment (e.g., `execute move(north)`)
- `achieve`: set new subgoal that triggers another plan (e.g., `achieve reach(rock)`)
- `add`: add new belief to belief base (e.g., `add visited(current_location)`)
- `remove`: remove existing belief (e.g., `remove obstacle(north)`)
- `update`: modify existing belief (e.g., `update position(X, Y)`)

Always format plans as:
```yaml
EVENT: achieve event to be pursued
CONDITIONS:
  - condition to be satisfied
  - other conditions to be satisfied
OPERATIONS:
  - [execute|achieve|add|remove|update] operation to be performed
  - [execute|achieve|add|remove|update] other operations to be performed
```

Separate multiple plans with `---`. Use `<none>` for empty conditions or operations.
````

Listing 3.2: Fragment of the format restricting instructions that define the hybrid YAML and Prolog syntax that the LLM is expected to use to format its responses in the implemented PGP prototype.

3.5 Writing Generative Agent Specifications

When BDI agents are generative and can generate their own plans, the role of the programmer necessarily evolves to accommodate these new capabilities. Given the current state of technology, programmers remain essential for writing the initial plans of the agent and for providing hints about the intended semantics of the goals, beliefs, and actions they define. Writing semantic hints as part of transferring knowledge to the LLM for improved plan generation represents the primary focus of this section. The aim is to show how developers can provide additional domain knowledge to the LLM in an easily maintainable way, alongside the agent's specification, addressing item **(RQ3)**.

In the general case, it is not guaranteed that LLMs will be able to understand the intended meaning of the goals, beliefs, and actions they are asked to generate plans for/with. This is because there is no guarantee that BDI programmers would use intuitive and self-explanatory names for them, or, that the LLM is trained on the BDI programming language of choice. For this reason, the encoding procedure (§ 3.4.2) is designed to include some natural-language description of each aspect of the agent's internal state.

Yet, descriptions cannot be automatically generated—as any automatic procedure may risk being inaccurate or incomplete w.r.t. the programmer's intent—and programmers would be better writing them directly. Descriptions should be written in natural language and should explain the meaning of both the *admissible* and *actual* goals, beliefs, and actions each agent may have.

Accordingly, a PGP-equipped BDI agent requires agent programs to be written in a way that *(i)* natural-language descriptions for admissible and actual mental abstractions are part of the agent's code, and *(ii)* those descriptions are used to generate the prompt for the PGP. To this end, the AgentSpeak(L) framework, and consequently BDI programming languages, should be extended with ad-hoc syntactical constructs for tagging cognitive abstractions with their intended meaning. Furthermore, to minimize the programmer's burden, these constructs should be optional, and allow for templating and reusing descriptions. Section 4.4.4 introduces an implementation of these constructs and listing 3.3 shows an example of application to document an admissible goal, using a syntax that will be detailed in section 4.4.

```
goals {
   admissible {
      +achieve("reach"("Object")).meaning {
         "reach a situation where ${args[0]} is in the position of the agent" +
            " (i.e. there_is(${args[0]}, here))"
      }
   }
}
```

Listing 3.3: An example of how a description enriches an admissible goal with an explanation of its meaning, to the benefit of both the human programmer and the LLM-based PGP.

Chapter 4

Implementation

Building on the conceptual framework introduced in chapter 3, this chapter presents its concrete implementation within JaKtA. The prototype supports both on-demand and reactive PGPs, extending the existing AgentSpeak(L)-based BDI engine and its Kotlin-based DSL for agent programming.

The implementation consists of several components: an integration layer that connects the generative process with the BDI engine (§4.1); a logging system designed to facilitate debugging and empirical evaluation (§4.2); a modular pipeline architecture for defining generative processes (§4.3); and a generative agent specification that enables configuration and customization of PGPs (§4.4).

4.1 Integration with the BDI Engine

This section describes how the reactive PGP has been integrated into the existing BDI engine of JaKtA, following the structure previously outlined (§ 3.1). The integration focuses on two main aspects: first, a set of interfaces defines the contract through which the BDI engine can invoke generation procedures and use their results (§ 4.1.1); second, the `GenerationManager` acts as the main entry point, allowing the BDI engine to interact with a PGP (§ 4.1.2).

Figure 4.1 illustrates the resulting augmented control cycle of a BDI agent. The generative process is triggered when an agent fails to retrieve a relevant plan. If the generation succeeds, the resulting plans are added to the plan library and

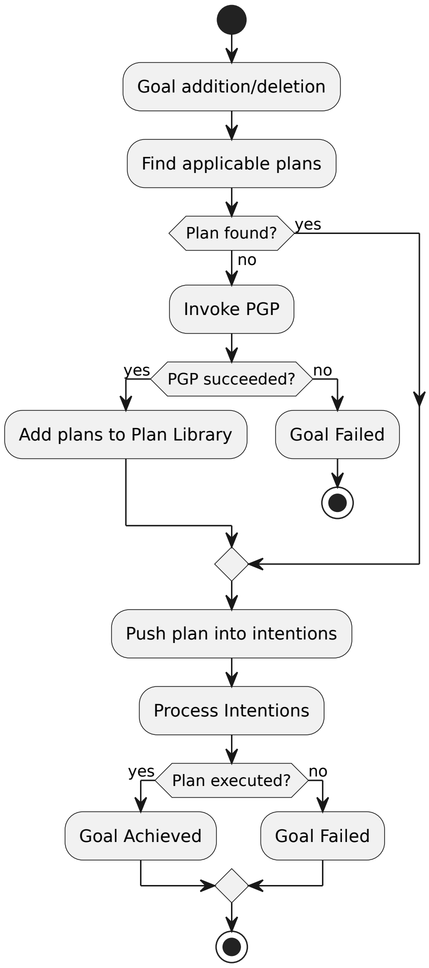


Figure 4.1: Simplified BDI control cycle that uses a reactive PGP to dynamically generate new plans at runtime.

may be pushed into the intention stack. If generation fails, the goal is marked as failed, preserving the fallback behavior of traditional BDI agents.

4.1.1 The PGP contract

The contract is built around five fundamental interfaces.

The `GenerationConfig` interface encapsulates all configuration parameters needed for plan generation, allowing different generation approaches to specify their requirements without coupling to specific implementations.

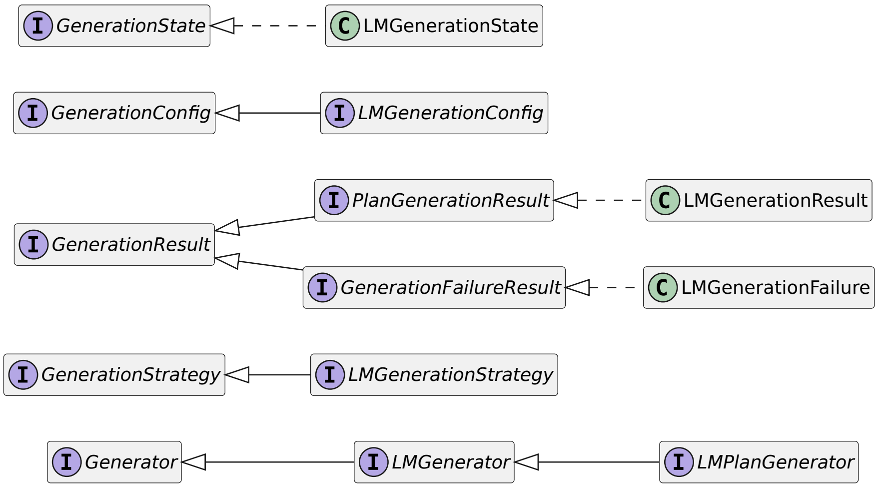


Figure 4.2: The interfaces that define the contract for plan generation exposed by the BDI engine and the interfaces and classes that implement this contract for a LM-based PGP.

The `GenerationResult` interface standardizes how generation outcomes are communicated back to the BDI engine, whether successful or failed.

The `Generator` interface defines the core generation capability, abstracting away the specific mechanisms used to produce plans.

The `GenerationState` interface maintains the context throughout the generation process, tracking the plan generation goal, the associated PGP identifier, and logging capabilities.

The `GenerationStrategy` interface binds together a specific generator with its configuration. This design allows the BDI engine to work with different generation approaches transparently, regardless of the kind of planners used and of how they are configured.

The LM-prefixed interfaces and classes, shown in fig. 4.2, implement the language model-based PGP.

The `LMGenerationConfig` extends the base configuration with LM-specific parameters including model identifiers, temperature settings, maximum token limits, and server endpoints for API communication. In table 4.1 an outline of the configuration that the user can provide is given.

The LM-based implementation provides two concrete types of generation re-

Parameter	Type	Description
`model`	`String`	Specifies the language model to be used for plan generation (e.g., GPT-4, Claude, etc.)
`temperature`	`Double`	Controls the randomness of the generated output. Lower values produce more deterministic results, higher values increase creativity
`maxTokens`	`Int`	Maximum number of tokens that can be generated in the response, limiting the length of generated plans
`url`	`String`	The endpoint URL for the language model API service
`token`	`String`	Authentication token required to access the language model API
`requestTimeout`	`Duration`	Maximum time to wait for a complete API request-response cycle
`connectTimeout`	`Duration`	Maximum time to wait when establishing a connection to the API service
`socketTimeout`	`Duration`	Maximum time to wait for data transfer over an established connection
`contextFilter`	`ContextFilter`	Filter component that determines which contextual information should be included in the generation prompt
`promptBuilder`	`PromptBuilder`	Component responsible for constructing the prompt sent to the language model, including context formatting and instruction generation

Table 4.1: Configurable Parameters for the PGP.

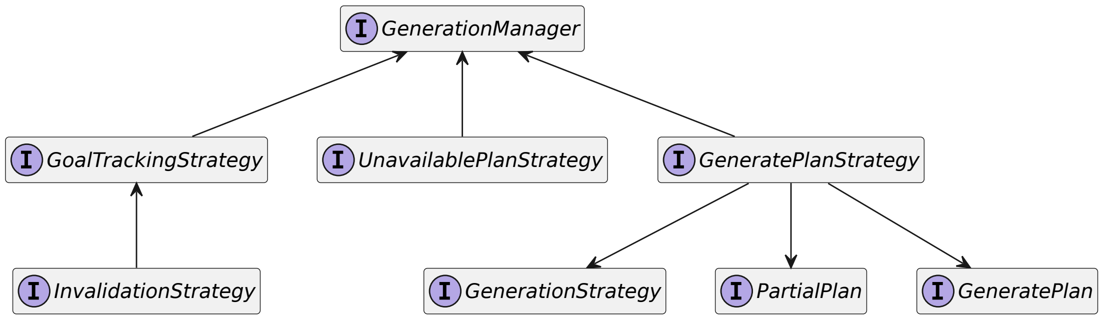


Figure 4.3: Components of the `GenerationManager`, which orchestrates the PGP.

sults: `LMGenerationResult` for successful generations and `LMGenerationFailure` for error cases. The success case handles both the generated plans and the admissible goals and beliefs invented by the LLM, along with natural language descriptions.

The `LMGenerationState` extends the base state interface with chat history management. This supports iterative refinement scenarios where the initial generation may be incomplete or require clarification.

The `LMPlanGenerator` implements the core generation logic through two key components: a `RequestHandler` responsible for sending requests to the language model according to the chosen provider, and a `Parser` that interprets the model's responses and extracts structured plan representations. This separation allows for independent evolution of the communication protocol and of the parsing logic.

The `LMGenerationStrategy` binds together the LM-specific generator and configuration.

4.1.2 The Generation Manager

The `GenerationManager` is the central component responsible for orchestrating the PGP within the agent's control cycle. Its role spans across initiating plan generation, tracking execution outcomes, managing failures, and maintaining the integrity of the dynamically generated plan library, delegating responsibilities to specialized strategy implementations. Figure 4.3 provides an overview of the core components and their interactions, which will be further discussed in the following sections.

Generation Goals

The `GeneratePlan` goal serves as the primary interface between the BDI control cycle and the plan generation process. This goal encapsulates a request for plan generation, storing both the target goal for which plans should be generated and optional configuration parameters that control the generation process. When a `GeneratePlan` goal is encountered during the execution of an intention, it is processed by the `GenerationManager`. The system first checks if a generation strategy is available—if not, it returns a failure feedback indicating that plan generation cannot proceed without a configured strategy. This design ensures that agents can operate with or without plan generation capabilities, providing backward compatibility with traditional BDI systems.

The execution of a `GeneratePlan` goal involves several steps: *(i)* the generation strategy is optionally updated with the provided configuration, *(ii)* a generation state is initialized with the current agent context, and finally *(iii)* the actual plan generation request is made to the generation strategy. This process integrates with the existing BDI control cycle, allowing generated plans to be available in the plan library when the generation completes.

Partial Plans

Plans generated by the PGP are stored as partial plans, which maintain references to their generation context and can be dynamically invalidated by using a plan invalidation mechanism. Unlike traditional plans in BDI systems, partial plans hold information about their origin, specifically the generation goal that created them and the configuration used during generation. When a partial plan is scheduled in an intention, only the goals being tracked are included if there are any, otherwise all the goals are used to create an activation record as in standard BDI plans. When determining if a partial plan is applicable to a given event, the system checks the standard trigger matching and guard conditions ignoring the source of the beliefs [1].

[1] Agents that employ a generation strategy currently ignore belief sources when unifying with the belief base. This limitation exists because sources cannot be reliably added to generated beliefs automatically, and the LLM is not prompted to annotate beliefs with sources to avoid increasing prompt complexity.

Tracking the Generated Goals

The system implements a goal tracking mechanism to monitor the execution of goals within generated partial plans. This is achieved through the `TrackGoalExecution` goal type, which wraps the actual goals that need to be executed and provides hooks for verifying their success or failure. The tracking mechanism serves multiple purposes: it enables the system to detect when generated plans fail to achieve their intended outcomes, and it provides feedback that is used for debugging and that can potentially be used to improve future plan generation.

The `GoalTrackingStrategy` manages the tracked goals, replacing them with their actual counterparts during execution and checking their results. When a tracked goal executes successfully, the system updates the plan library to "untrack" the goal. When all tracking goals of a partial plan get removed, then the generated plan can be considered as successfully executable. This of course does not mean that the plan achieves the goal for which it was generated, since no verification mechanism is in place, nor that it does so optimally. The tracking mechanism also handles failure cases by triggering the invalidation strategy when tracked goals fail. This ensures that plans that prove ineffective in practice are removed from the plan library, preventing the agent from repeatedly attempting unsuccessful approaches. In principle this can also be used to trigger a plan repair mechanism since it also provides the execution feedback of the failed goal being tracked.

Handling Plan Unavailability

The PGP trigger strategy determines when plan generation should be initiated during the control cycle. The system supports both the on demand and reactive approaches described in section 3.2, where plan generation is triggered by the `generate_plan` action or when the agent encounters situations for which no applicable plans exist. The `UnavailablePlanStrategy` interface intercepts plan selection failures and evaluates whether they can be resolved through dynamic plan generation. The system distinguishes between the absence of relevant plans and the presence of relevant but inapplicable plans. The `UnavailablePlanStrategyImpl` implements this logic by first categorizing the type of unavailability and then applying appropriate resolution strategies. When no plans are found for a given event trigger, the system checks if the event represents an achievement goal. If so, it creates a new p_{genex} plan (§ 3.2.2) using the `GenerationPlanBuilder`, adding it to the plan library along with the `missing_plan_for(G)` belief and an event to trigger the generation process. For cases where relevant plans exist, but none are applicable due to failed preconditions, the system provides detailed feedback about the applicability failures. This includes logging information about which guards failed and why, enabling developers and potentially the agent itself to understand the reasons for plan rejection.

The Plan Invalidation Mechanism

The plan invalidation mechanism has the responsibility of maintaining the integrity of the generated plan library by removing plans that have proven ineffective. The current `InvalidationStrategy` implements this mechanism by selectively removing partial plans from the plan library when they are associated with failed goal executions. The system specifically targets partial plans for removal, preserving manually authored plans and cleaning up dynamically generated content that has proven problematic.

There are no mechanisms in place to prevent unchecked growth of the plan library. To prevent this issue, an `InvalidationStrategy` could also include mechanisms to "forget" plans if they are not used over time, to rank them and periodically remove the least important ones, to factorize them or to offload them to an external storage once a threshold is reached.

Plan Generation Strategy

Figure 4.4 shows an overview of the `GeneratePlanStrategy` interface. It handles the initial setup and delegates the actual generation request to a chosen strategy. It then uses the `GenerationResultBuilder` interface to construct `PlanGenerationResult`s from successful plan generations, that can be integrated transparently into the BDI engine. Each of the three `Updater`s within the result builder handles a specific aspect of integrating the generated content into the agent's context.

The `TemplateUpdater` class manages the integration of newly generated ad-

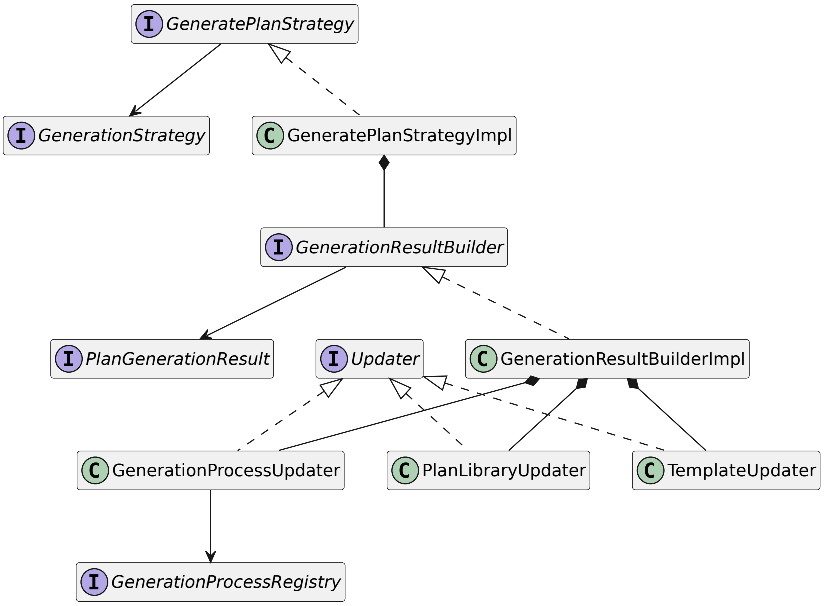


Figure 4.4: Components of the `GeneratePlanStrategy`, which invokes the generation strategy provided by the user and updates the result to the agent that requested the generation.

missible beliefs and goals into the agent's context. For admissible beliefs, it merges the existing beliefs with newly generated ones, using the rule head functor as a unique identifier to prevent duplicates. Analogously, for admissible goals, it combines existing and new goals using the trigger value functor to distinguish them.

The `PlanLibraryUpdater` handles the integration of newly generated plans into the agent's plan library. It processes each generated plan by wrapping the plan goals with a `TrackGoalExecution`. The updated plans are then merged with the existing plan library. Newer plans overwrite existing ones with the same trigger and context.

The `GenerationProcessUpdater` manages a registry of generation processes, updating their state according to generation results. This underlying `GenerationProcessRegistry` is implemented as a map, linking each goal generation request to its corresponding generation state. The use of a registry potentially allows handling long-running PGPs, including those involving LLMs that may require multiple interaction turns.

4.2 Logging System

As part of the development of the PGP, a logging system was introduced in JaKtA to ease debugging and to extract structured data from execution traces as part of the experimental evaluation. The logging system is based on the interfaces `LogEventContext` and `LogEvent`. The first interface acts as a wrapper of `LogEvent`s and provides information used to unequivocally identify each event with the identifier of the MAS execution, the identifier of the agent and the identifier of the running PGP, if they are relevant for the event. `LogEvent` is the base interface from which all other log events extend. In fig. 4.5 the base hierarchy is shown. There are three main kinds of events:

- `AgentEvent` references log events generated by each agent, relative to its internal mechanics and its data structures. For example, this includes events related to the selection, addition or removal of intentions, plans or events. In fig. 4.5 and fig. 4.6 all the logged events are shown.

- `EnvironmentEvent` references the effects that are applied to the environment after the execution of the control cycle of an agent. This includes for example receiving, sending or broadcasting messages. In fig. 4.7 all the log events relative to the environment provided by the BDI engine are shown. By extending the interface, new kind of environmental conditions can be logged from user-provided environments.

- `ExecutionFeedback` references events relative to the result of the execution of the sense-reason-act cycle. This includes whether relevant or applicable plans are not found during the deliberation or if the execution of an action fails in the action phase. In fig. 4.8 and fig. 4.9 all the feedback events provided by the BDI engine are shown. By extending the interface, the user can record new kinds of feedback as a result of the execution of custom actions.

The `JaktaLogger` interface provides the starting point from which the loggers are defined, and provides a set of functions to log strings or objects, according

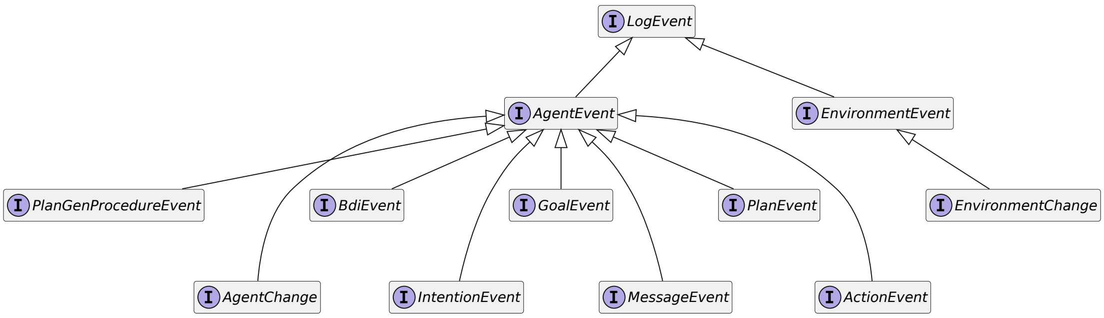


Figure 4.5: `LogEvent` interface hierarchy.

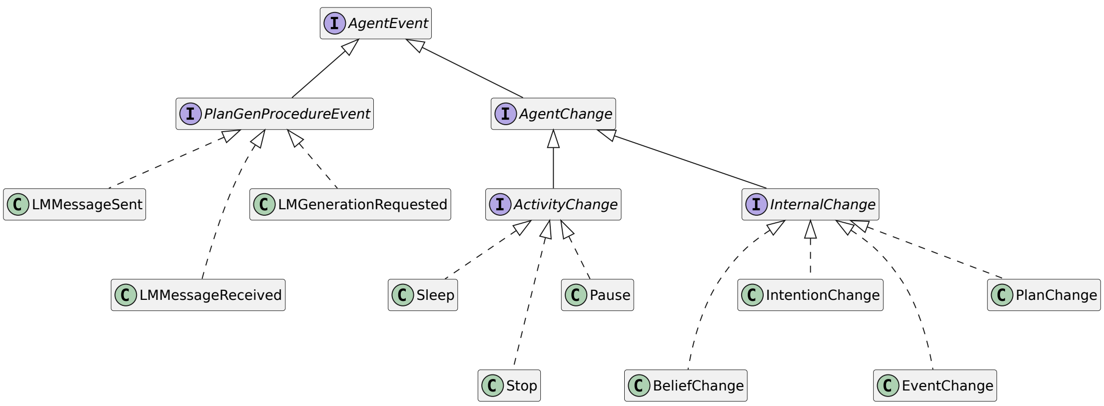


Figure 4.6: `AgentEvent` interface hierarchy.

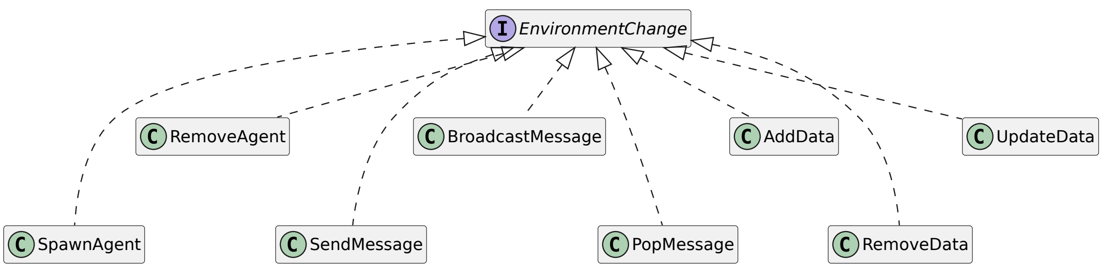


Figure 4.7: `EnvironmentChange` interface hierarchy.

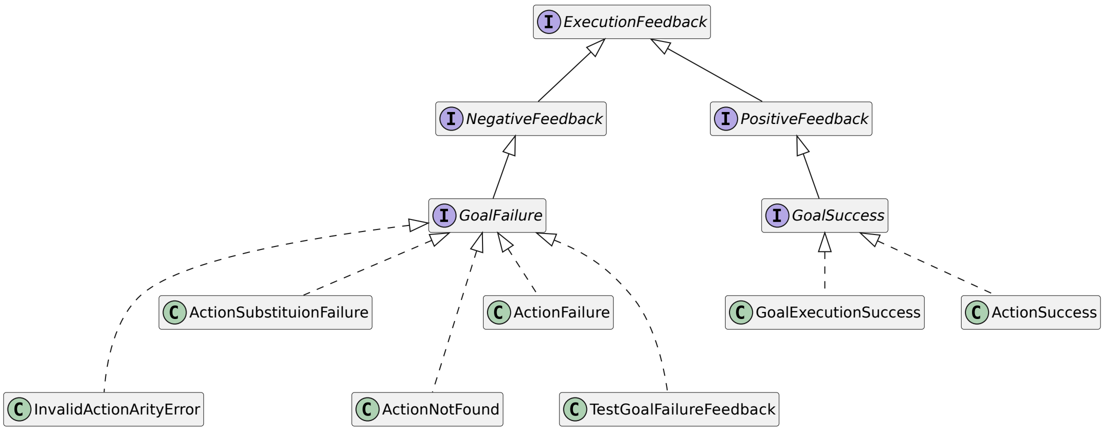


Figure 4.8: `ExecutionFeedback` interface hierarchy considering only goals' results.

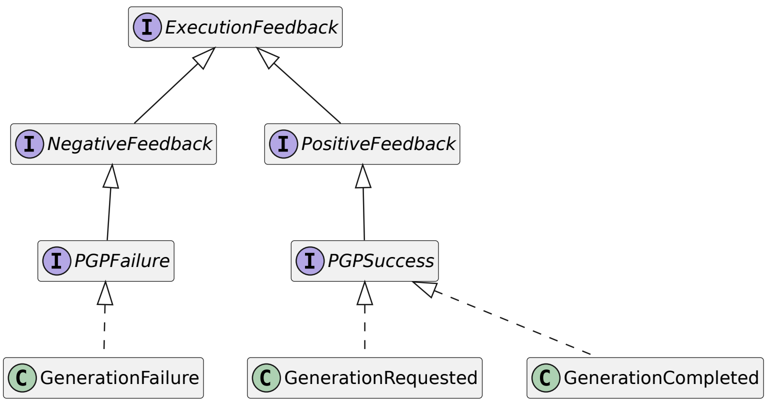


Figure 4.9: `ExecutionFeedback` interface hierarchy considering only PGPs' results.

to the chosen log level. It is implemented by the `MasLogger`, `AgentLogger` and `PgpLogger` interfaces.

When logging is enabled, a `MasLogger` instance is created during the initialization of the MAS. Subsequently, each agent is assigned its own `AgentLogger`. A `PgpLogger` is instantiated by the generation manager when a new generation state is initialized, specifically during the processing of a `GeneratePlan` goal.

The `MasLogger` is responsible for logging `EnvironmentEvents`, which include environmental effects and user-defined environment events. The `AgentLogger` logs all `AgentEvents` except those of type `PlanGenProcedureEvent`, which are handled exclusively by the `PgpLogger`.

Both `MasLogger` and `AgentLogger` are also responsible for logging events of type `ExecutionFeedback`.

Logging behavior is governed by the `LoggingConfig` class, which stores configuration flags such as whether to log to the console, to one or more files, or to a remote server. If a MAS is created without this configuration, then the application will not log anything.

4.3 Generative Process Pipeline

This section presents the five core components that are used to implement LLM-based generation strategies. *Context filters* (§ 4.3.1) retain relevant knowledge, *formatters* (§ 4.3.2) encode this knowledge into LLM prompts, and *parsers* (§ 4.3.3) decode structured responses. *Request handlers* (§ 4.3.4) manage LLM interactions, while *plan generators* (§ 4.3.5) orchestrate the overall workflow.

4.3.1 Context filters

A context filter implements the interface `ContextFilter`, which takes in input an `ExtendedAgentContext` and returns a filtered `ExtendedAgentContext`. The extended agent context includes both the agent context and all the additional data that might be used to build the prompt of a LLM, such as the external actions provided by the environment or the initial goal provided by the user. In section 4.4.1 an example filter definition is shown. Context filters control what information reaches the language model, reducing information overload and improving response quality.

4.3.2 Formatters

A formatter implements the `Formatter` interface and transforms internal data structures into human-readable string representations for LLM consumption, logging and debugging. The system includes built-in formatters for encoding agent and environment context in LLM prompts. As an example, listing 4.1 shows how the action formatter transforms action objects into descriptive strings that include the action signature, parameter names, and optional purpose description.

```kotlin
val actionsFormatter = Formatter<Action<*, *, *>> { action ->
    buildString {
        val signature = action.actionSignature
        append(signature.name, "(")
        append(
            signature.parameterNames.takeIf { it.isNotEmpty() }
                ?.joinToString { it.capitalize() }
                ?: (1..signature.arity).joinToString { "Parameter$it" }
        )
        append(")")
        action.purpose?.let { append(": $it") }
    }.dropWordsWithTrailingNumbers()
}
```

Listing 4.1: Implementation of the action formatter that converts `Actions` into human-readable string representations for LLM prompts.

4.3.3 Parsers

A parser implements the `Parser` interface and converts the text generated by the LLM into structured internal representations. The parser implemented for the prototype extracts YAML code blocks from input text. It attempts to decode content as either `PlanData` or lists of `TemplateData`. Its responsibilities include processing different template types (beliefs and goals), validating the parsed structures, and accumulating any parsing errors. Figure 4.10 shows both the type of errors that can be accumulated by the parser and the kind of results it can return. The final result aggregates successfully parsed plans, admissible beliefs, and admissible goals

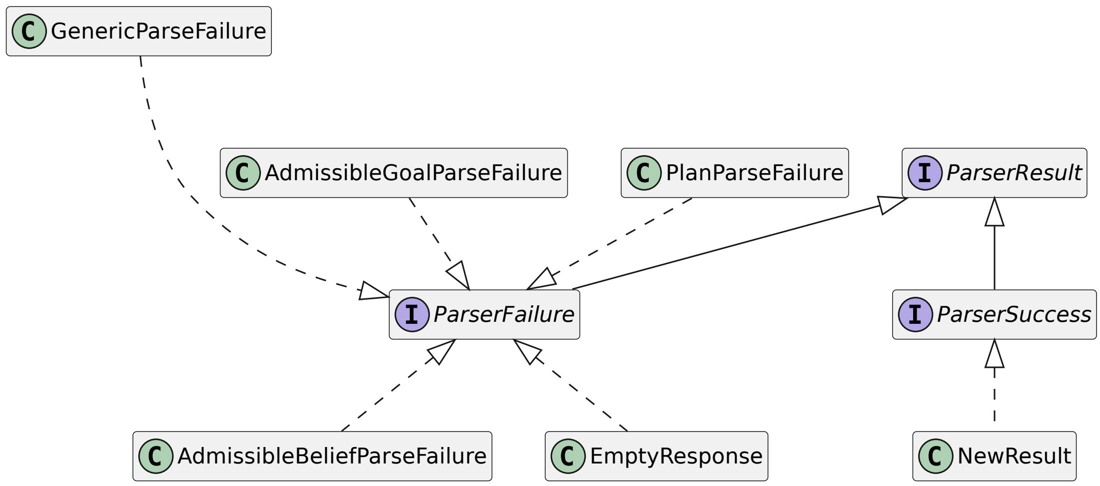


Figure 4.10: Set of interfaces that handle the parser's results.

in the `NewResult` class, along with parsing errors, if there are any, or returns an `EmptyResponse` if no valid content was extracted.

4.3.4 Request Handlers

The `RequestHandler` interface encapsulates the configuration, communication, and delegation needed to carry out text-based completions with LLMs, returning structured results that the rest of the pipeline can consume.

A text generation request is built from the current generation configuration and state. Once the request is ready, it is sent to a `RequestProcessor`. The `RequestProcessor` handles low-level communication with the model through an OpenAI-compliant API and passes the raw response through a `Parser` and translates the `ParserResult` into a structured `RequestResult`.

The `RequestResult` interface represents whether a request had success or not. If the request is successful a `NewRequestResult` object is created, which encapsulates parsed plans (`NewPlan`), admissible goals, beliefs, and any potential parsing errors in the `NewResult` object. A `NetworkRequestFailure` is returned instead if there are problems such as timeouts or invalid responses.

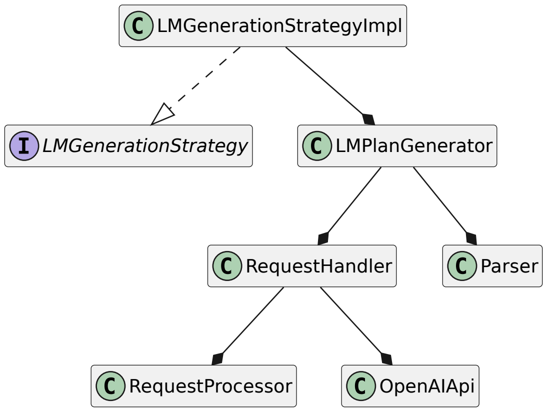


Figure 4.11: Set of classes that compose a LLM-based plan generation strategy.

4.3.5 Plan Generators

As it is shown in fig. 4.11 the `LMGenerator` interface defines the high-level abstraction that is used by the `LMGenerationStrategy` to handle generation requests. It relies on the `RequestHandler` to issue requests and parse results, while also logging relevant messages and updating the generation state.

4.4 Generative Agent Specification

This section describes how to declare and configure generation strategies for use by the PGP, building upon the components introduced in the previous section. The configuration process is structured in several areas. Prompt templates can be defined using a Kotlin DSL (§ 4.4.2), while the scope and parameters that define the generation strategy are selected through dedicated configuration mechanisms (§ 4.4.3). For enhanced performance, custom filters serve to regulate information flow to the LLM (§ 4.4.2), and natural language documentation provides helpful operational hints (§ 4.4.4). The specification of on-demand generation goals offers additional flexibility (§ 4.4.5), and custom log events allow to debug the generation process and support automatic evaluation procedures (§ 4.4.6).

4.4.1 Implementing Custom Filters

As the size of the belief base of an agent, its set of actions and available plans grows, it becomes impractical and performance-degrading to keep all that information in the context. For that reason, mechanisms to filter out information are needed. Custom filters provide a mechanism to control what information from the BDI agent state is presented to the LLM during plan generation.

The filtering mechanism operates on an `ExtendedAgentContext` structure that encapsulates the complete agent state, including the initial goal, the current context with beliefs and plans, and available external actions.

Effective filtering significantly improves the performance of LLM-based BDI agents by: *(i)* reducing the token count in prompts, leading to faster response times and lower computational costs, *(ii)* decreasing the cognitive load on the LLM by presenting only relevant information, and *(iii)* minimizing the risk of the LLM being distracted by irrelevant or contradictory information.

When implementing custom filters, the trade-off between information completeness and performance must be carefully balanced. Overly aggressive filtering may remove crucial information that could lead to better plans, while insufficient filtering may degrade performance without providing meaningful benefits.

```
val metaPlanFilter = ContextFilter { extendedContext ->
   extendedContext.createNewTriggerFromGoal()?.let { trigger ->
      val generationPlanId = GenerationPlanBuilder.getGenerationPlanID(trigger)
      extendedContext.copyWithFilteredPlans { plan ->
         plan.id != generationPlanId &&
               plan.trigger !is TestGoalFailure &&
               plan.trigger !is AchievementGoalFailure
      }
   } ?: extendedContext
}
```

Listing 4.2: Default context filter implementation that removes meta-plans and failure-handling plans to streamline the LLM context for plan generation.

The built-in filter, shown in listing 4.2, removes the special generation plans that are automatically created when no applicable plan exists for a goal, as described in section 3.2.2. These meta-plans are internal mechanisms that would confuse the LLM during plan generation and are thus omitted.

4.4.2 Defining Prompt Builders

The PGP requires incorporating dynamic content, structured data, and contextual information. Traditional string concatenation approaches become unwieldy and error-prone when dealing with prompts containing nested hierarchies and conditional content. A Kotlin DSL was developed to address this challenge and enable declarative prompt construction while maintaining type safety and readability. In listing 4.3 an example of use of the DSL is shown.

The `PromptScope` class serves as the primary builder for prompt construction. It maintains an internal list of `PromptSection` objects and provides methods for adding different types of content. This design separates content from presentation, allowing the same section to be rendered at different heading levels depending on its position in the hierarchy.

The DSL provides several directives for adding content to user or system prompts: *(i)* `section`: creates a titled section with nested content, *(ii)* `fromFile`: includes content from resource files, *(iii)* `fromFormatter`: applies formatting functions to data, and *(iv)* `fromString`: adds literal text content. Providing both user and system DSLs reflects how LLMs differentiate between system instructions and user queries and allows building better prompts. This separation of concerns improves code clarity, prevents accidental misuse of the two roles, and aligns with best practices in prompt engineering.

4.4.3 Declaring the Generation Strategy

One of the most relevant aspects of the specification involves selecting an appropriate generation strategy and defining its scope and configuration parameters. The framework implements a LLM-based generation strategy which leverages the pipeline components detailed in section 4.3. After selecting a generation strategy and configuring its parameters, the scope where the strategy applies can be defined. The system supports three hierarchical declaration levels, each offering different application scopes: *(i)* MAS-Level: defines the default generation strategy for the entire Multi-Agent System (listing 4.4), *(ii)* agent-level: specifies generation strategies for individual agents, overriding MAS-level settings (listing 4.5), and *(iii)* plan-level: sets generation strategies for specific plans, taking precedence over

```
system("SystemPrompt") {
    section("System Message") {
        fromFile("system.md")
    }
}

user("UserPromptWithHints") { ctx ->
    section("User Message") {
        fromString("Below is your internal state and the specific goal I need you to plan for.")
        section("Agent's internal state") {
            section("Beliefs") {
                section("Admissible beliefs") {
                    fromFormatter(ctx.admissibleBeliefs) {
                        formatAsBulletList(it, admissibleBeliefsFormatter::format)
                    }
                }
                section("Actual beliefs") {
                    fromFormatter(ctx.beliefs.asIterable().toList()) {
                        formatAsBulletList(it, beliefsFormatter::format)
                    }
                }
            }
            section("Goals") {
                section("Admissible goals") {
                    fromFormatter(ctx.admissibleGoals) {
                        formatAsBulletList(it, admissibleGoalsFormatter::format)
                    }
                }
                section("Actual goals") {
                    fromFormatter(ctx.goals) { plans ->
                        val triggers = plans.map { it.trigger }
                        formatAsBulletList(triggers, triggerFormatter::format)
                    }
                }
            }
            section("Admissible actions") {
                fromFormatter(ctx.internalActions + ctx.externalActions) {
                    formatAsBulletList(it, actionsFormatter::format)
                }
            }
        }
        section("Expected outcome") {
            val formattedGoal = goalFormatter.format(ctx.initialGoal.goal)
            fromString("Create plans to pursue the goal: $formattedGoal.")
            fromString(
                """
                Output only the final set of plans with no alternatives or intermediate attempts.
                End with an additional YAML block that contains a list of any new admissible goals
                    and beliefs you invented, including their natural language interpretation.
                """.trimIndent(),
            )
        }
    }
}
```

Listing 4.3: Example of a user prompt template, which is populated at runtime according to the state of the agent and the content of static files.

both agent and MAS-level configurations (listing 4.6). This hierarchical structure operates on a specificity-based override system. The precedence flows from plan-level (the highest priority) to agent-level (medium priority) to MAS-level (the lowest priority), to make sure that the most specific configuration always governs the behavior. When conflicts arise between different declaration levels, the system employs a granular resolution mechanism. Rather than replacing entire configuration blocks, individual parameters are overwritten based on the hierarchy. This allows for fine-grained control where, for example, an agent might inherit most parameters from the MAS level while only modifying specific settings such as temperature or max tokens.

```
mas {
    lmGeneration {
        connectTimeout = 2.seconds
    }
    agent("Printer") { ... }
}
```

Listing 4.4: Example of a generation strategy declared at the MAS level.

```
mas {
    agent("Printer") {
        lmGeneration {
            connectTimeout = 2.seconds
        }
        goals { ... }
        plans { ... }
    }
}
```

Listing 4.5: Example of a generation strategy declared at the agent level.

4.4.4 Writing the Documentation

The DSL provided by JaKtA has been extended with additional syntactical constructs, in order to allow programmers to provide hints, as explained in section 3.5. The new entries of the DSL are: *(i)* the `.meaning{…}` blocks, which can be used to tag *actual* goals, beliefs, or actions with their intended meaning, via a postfix syntax; and *(ii)* the `admissible{…}` blocks, which can be used to tag *admissible*

```
mas {
   agent("Printer") {
      goals { ... }
      plans {
         +achieve("print_numbers"(X, Y)) then {
            ...
         } givenLMConfig {
            connectTimeout = 2.seconds
         }
      }
   }
}
```

Listing 4.6: Example of a generation strategy declared at the plan level.

goals or beliefs with their intended meaning, via a prefix syntax. In both cases, the meaning is expressed as a string, attained via string interpolation, which in turn may include the functor and the arguments. The strings are then exploited by the PGP to generate the prompt for the LLM through formatters.

An example of use of the new constructs is shown in listing 4.7.

```
agent("ExplorerBot") {
  goals { ... }
  beliefs {
     admissible {
        +fact { "obstacle"("Direction") }.meaning {
           "there is an $functor to the ${args[0]}"
        }
     }
  }
}
```

Listing 4.7: Example of a JaKtA agent extended with natural-language descriptions

Moreover, the `remark` method can be used to provide the LLM with additional constraints or suggestions to further guide the generative process. The additional instructions are added to the prompt in a dedicated section.

4.4.5 Specifying on-demand Generation Goals

In listing 4.8 an example of agent specification that explicitly uses the on-demand PGP approach is shown. The agent has the goal of printing the numbers from zero to ten, but instead of providing a pre-written plan with specific implementation details, the plan body contains only a call to the `generate_plan` action. This provides an example of how programmers can delegate the actual plan generation to the underlying LLM while maintaining control over when and under what conditions the generation should occur. When the agent needs to achieve the goal `print_numbers(0, 10)`, it will execute the corresponding plan, which triggers the PGP to generate an appropriate implementation plan. The configured LLM will receive the natural language description "Print the numbers from 0 to 10" along with the specific context and generate a concrete plan that implements the required functionality.

```
mas {
    lmGeneration {
        model = "openai/gpt-4.1"
        temperature = 0.5
        maxTokens = 1024
    }
    agent("Printer") {
        goals {
            +achieve("print"(0, 10))
            admissible {
                +achieve("print_numbers"("start", "end")).meaning {
                    "Print the numbers from ${args[0]} to ${args[1]}"
                }
            }
        }
        plans {
            +achieve("print"(X, Y)) then {
                generatePlan("print_numbers"(X, Y))
            }
        }
    }
}
```

Listing 4.8: Example of a JaKtA agent that will try to generate a plan to print the numbers from zero to ten using the provided generation strategy.

4.4.6 Defining Custom Log Events

While the logging system described in section 4.2 covers the standard BDI engine operations, applications might require domain-specific logging capabilities that capture the unique characteristics of their problem domain. Custom environments that extend the `Environment` interface can benefit from implementing and logging domain-specific events that capture when the system reaches particular states of interest. These specialized events provide more granular insight into the environment's behavior than generic state transitions. Analogously, when creating custom actions, the `addFeedback` method from the `Action` interface enables the provision of detailed, domain-specific `ExecutionFeedback` log events that can provide richer diagnostic information of action outcomes beyond simple binary success or failure indicators.

```
externalAction("move", "direction") {
    val env = environment as? GridWorldEnvironment
    if (env != null) {
        val updatedEnvState = env.parseAction(actionName)
        val oldPosition = env.data.state()?.agentPosition
        if (updatedEnvState != null && oldPosition != null) {
            updateData("state" to updatedEnvState)
            val newPosition = updatedEnvState.agentPosition
            val feedback = updatedEnvState.objectsPosition.entries
                .find { it.value == newPosition }
                ?.let { ObjectReachedEvent(it.key, arguments) }
                ?: MoveActionSuccess(oldPosition, newPosition, arguments)
            addFeedback(feedback)
        }
    }
}
```

Listing 4.9: Implementation of the `move` action, which provides a more specific feedback than the default `GenericActionSuccess` when the action completes successfully.

These user-provided log events serve multiple purposes: they streamline experimentation workflows, support monitoring and debugging capabilities, and can provide structured input to aid LLMs in plan repair and refinement. As an example the `move` action, which is shown in listing 4.9, provides specific feedback based on movement outcomes, rather than returning generic success messages. When the action executes successfully, it returns a `MoveActionSuccess` object containing movement details. When an agent reaches its home location, the system generates an `ObjectReachedEvent` providing feedback about this outcome. The feedback of the action is used during the evaluation of the explorer bot example in chapter 5 to assess whether a PGP completes the requested goal, by examining the execution trace for the presence or absence of the `ObjectReachedEvent`.

Chapter 5

Evaluation

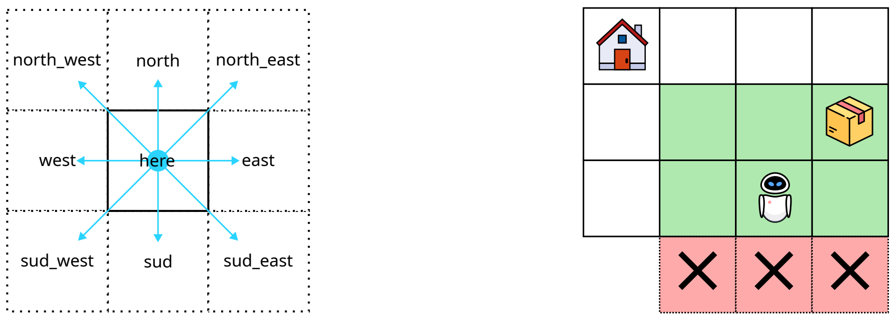
This chapter evaluates the reactive PGP implementation within the "Explorer Robot" domain, defined in section 5.1. The agent-based application used for the evaluation features specifications enhanced by natural language descriptions, first conceptualized in section 3.5 and then implemented in section 4.4. The experimental methodology employed to assess the agent's behavior is presented in section 5.2 and the resulting outcomes are discussed in section 5.3.

5.1 The explorer robot application

In the explorer robot domain, an agent acts as an explorer (§ 5.1.1) and has the objective of reaching home within a bidimensional grid world environment (§ 5.1.2), where it can perceive obstacles in its immediate surroundings and move in the eight cardinal directions. Figure 5.1 shows the environment model.

5.1.1 Agent

The `ExplorerRobot` agent, whose specification is shown in listing 5.1, is designed to achieve high-level goals such as reaching a target object (e.g., "home"). It declares a set of admissible goals and beliefs that are enhanced with natural language explanations.



(a) Nine admissible directions

(b) Example of environment model

Figure 5.1: Environment modelling for the explorer robot domain. The agent is in discrete bidimensional space, where it can move in eight admissible directions, namely: the cardinal ones, with north pointing up. Direction `here` denotes the current position of the agent.

```
agent("ExplorerRobot") {
  goals {
      admissible {
          +achieve("reach"("Object")).meaning {
              "reach a situation where ${args[0]} is in the position of the agent" +
                  " (i.e. there_is(${args[0]}, here))"
          }
      }
      +achieve("reach"("home"))
  }
  beliefs {
      admissible {
          +fact { "obstacle"("Direction") }.meaning {
              "there is an $functor to the ${args[0]}"
          }
          +fact { "there_is"("Object", "Direction") }.meaning {
              "there is an ${args[0]} in the given ${args[1]}"
          }
          +fact { "direction"("Direction") }.meaning {
              "${args[0]} is a direction"
          }
          +fact { "object"("Object") }.meaning {
              "${args[0]} is an object"
          }
      }
  }
}
```

Listing 5.1: The `ExplorerRobot` agent as implemented in JaKtA.

5.1.2 Environment

The application is based on a custom grid world environment, which is implemented by extending the `Environment` class provided by JaKtA. This environment provides two external actions for robot navigation. The `Move` action enables the robot to move in a specified direction, provided that no obstacles block the path. When executed, this action reports either the success of the movement or identifies any object that the robot encounters. If an obstacle prevents movement, the action has no effect. The `GetDirectionToMove` action assists with navigation by binding a provided variable to a randomly selected direction free of obstacles. If no directions are available for movement, the action produces no result. The environment generates percepts following the format shown in listing 5.2. The complete environment specification is detailed in listing 5.3. These percepts provide information about the robot's surroundings, including known directions and objects, obstacles that include grid borders, free cells, and the presence of objects in the eight cells that surround the agent's current position.

```
direction(north)
object(house)
object(box)
free(north).
free(north_west).
free(north_east).
free(west).
free(east).
obstacle(south).
obstacle(south_west).
obstacle(south_east).
there_is(box, north_east).
```

Listing 5.2: The set of percepts generated by the grid world environment.

5.2 Experimental Methodology

The evaluation employs the explorer robot application (§ 5.1) to assess the PGP capabilities with different sets of parameters. The experiment consists of a square grid world with five cells along each axis containing three static obstacles, with an explorer robot agent initially positioned at the center and a target `home` object

```
environment {
    from(GridWorld())
    actions {
        action(move).meaning {
            "move in the given ${args[0]}"
        }
        action(getDirectionToMove).meaning {
            "provides a ${args[0]} free of obstacles where the agent can then move"
        }
    }
}
```

Listing 5.3: Specification of the grid world environment in JaKtA.

located at one corner of the grid. The `home` object is positioned beyond the agent's initial perception range, and obstacles are placed to prevent direct navigation paths. The agent begins with the goal `reach(home)` but lacks any initial plan to achieve this objective.

Individual experiments are launched by an evaluation script (§ 5.2.1), using one of the selected language models (§ 5.2.2) configured with predetermined parameters (§ 5.2.3) to generate plans within a fixed timeout period. The evaluation framework measures both plan validity—whether the PGP successfully generates executable plans using available actions to reach the target—and plan quality, through metrics detailed in section 5.2.4.

5.2.1 Experimental Setup

Running LLMs demands substantial computational power, that puts them beyond the reach of typical consumer hardware. This reality has spawned the "LLM-as-a-Service" ecosystem, where providers offer model access through web APIs rather than local deployment. The landscape includes established players like OpenAI's Platform API and community-driven platforms like Hugging Face, alongside aggregators such as OpenRouter that provide unified access to multiple models from different providers. To conduct the experiments, the OpenRouter's API is selected as the primary interface since its unified endpoint structure eliminates the complexity of managing multiple provider-specific integrations, easing systematic comparisons across different models.

To assess the quality of generated plans, an automated evaluation procedure is devised based on the chosen APIs and evaluation metrics. A DVC [58] pipeline and Gradle tasks are used to automate the execution of a MAS for each experimental configuration to be tested and to store the results to a remote server.

The G-Eval [59] algorithm is used through the DeepEval [60] framework to evaluate a custom metric, the **Plan-Reference Alignment Score (PRAS)**. The metric is introduced to compare generated plans against the human baseline across three key dimensions: quality of abstraction, generalizability, and adherence to BDI principles. This evaluation approach is based on the LLM-as-Judge methodology [61], which allows to systematically rate plan quality using criteria that would be difficult to formalize through traditional metrics alone.

In order to have a reference ground truth for the evaluation, a human baseline is provided that represents the optimal plans to achieve the goal of reaching home, which are shown in listing 5.4.

```
plans {
    +achieve("reach"(Object)) onlyIf {
        "there_is"(Object, "here").fromPercept
    }
    +achieve("reach"(Object)) onlyIf {
        "there_is"(Object, Direction).fromPercept
    } then {
        execute("move"(Direction))
    }
    +achieve("reach"(Object)) onlyIf {
        not("there_is"(Object, `_`).fromPercept)
    } then {
        execute("getDirectionToMove"(Direction))
        execute("move"(Direction))
        achieve("reach"(Object))
    }
}
```

Listing 5.4: Set of baseline plans for the ExplorerRobot agent

5.2.2 Language Models

The experiments are based on these LLMs: `Gpt-4.1`, `Gemini 2.5 Flash`, `Deepseek Chat V3`, and `Claude Sonnet 4`. They are selected based on their state-of-the-art performance, as well as for their accessibility through public APIs with no exhaustivity claim.

5.2.3 Experiments' Parameters

The only parameters that were explicitly set are the temperature, the max tokens and the prompt type.

Temperature. The effect of temperature on plan generation was examined using three values: 0.1, 0.5, and 0.9. It was hypothesized that very low temperatures (0.1) would produce overly rigid outputs lacking the flexibility needed for novel objectives or ambiguous scenarios. Conversely, high temperatures (0.9) were anticipated to risk compromising output quality through incoherent or hallucinatory responses, as suggested by findings in [62].

Max Tokens. The maximum token limit of 2048 is chosen to ensure that the model can generate complete and contextually rich BDI plans. Shorter limits were avoided to risk premature truncation. Longer ones were not considered useful given the limited complexity of the task.

Prompt Type. Three distinct prompting strategies are evaluated to assess their impact on plan generation. The baseline approach excludes all natural language specifications, providing minimal context. The intermediate strategy keeps developer-provided hints and removes the remarks. The last approach preserves both hints and remarks, delivering the richest contextual information including environment-specific insights and action usage instructions. Hints and remarks were considered separately to assess how much different types of contextual information would influence the quality and effectiveness of the generated plans.

5.2.4 Evaluation Metrics

To evaluate the quality of generated plans, the metrics shown in table 5.1 are employed. Among these metrics, **GC** and **TSR** should be maximized (higher values are better), while **RR**, **GSA**, **BSA**, **GAT** should be minimized (lower values are better). The remaining metrics (**PC**, **CC**, **PBC**, **NGC**, and **NBC**) have no clear optimization direction, yet they provide valuable comparative insights, with both extremely low and high values potentially indicating issues. **PRAS** tends to one

Code	Metric	Description
PC	Plan Count	Total number of generated plans.
CC	Context Complexity	Average number of beliefs per plan context.
PBC	Plan Body Complexity	Average number of operations per plan body.
GC	Generalization Count	Number of generated plans using variables rather than constants.
RR	Redundancy Amount	Number of useless plans (e.g., subsumed).
NGC	Novel Goal Count	Number of newly-invented goals.
NBC	Novel Belief Count	Number of newly-invented beliefs.
GSA	Goal Semantic Alignment	Number of semantically-misaligned admissible goals.
BSA	Belief Semantic Alignment	Number of semantically-misaligned admissible beliefs.
TSR	Task Success Rate	Percentage of experiments where the agent achieves the `!reach(home)` goal.
GAT	Goal Achievement Time	Average steps required to reach the goal in successful runs.
PRAS	Plan-Alignment Reference Score	Alignment of the generated plans with the human reference

Table 5.1: Evaluation metrics for generated plans.

when the generated plans are similar to the reference ones, to zero when they are completely different.

```
"Extract invented goals, beliefs, and plans from the 'actual output'.",
"Compare extracted plans against 'expected output' plans for logical equivalence and coverage.",
"Assess if invented goals/beliefs are necessary or add needless complexity compared to 'expected
    output'.",
"Evaluate plan minimality; penalize unnecessary subgoals, conditions, or operations vs 'expected
    output'.",
"Verify that operations correctly use specified prefixes (execute, achieve, add, etc.) and
    admissible actions.",
"Check if conditions logically correspond to the intended plan activation scenario.",
"Score based on plan correctness, necessity of inventions, and adherence to minimality principle."
```

Listing 5.5: Prompt used for the LLM-as-judge evaluation.

5.3 Experimental Results

Each of the four chosen LLMs was queried ten times to account for the stochastic nature of LLM responses, for each of the three prompting configurations and for each of the three temperature values, yielding three hundred sixty total experimental runs.

The average amount of input tokens is 995, with a min of 808 and a max of 1196. The variability is due to presence of instances where multiple PGPs are invoked, so the total count is not always equal to the number of tokens of the initial input prompt, which has 1041 tokens. The average amount of output tokens is 223, with a min of 77 and a max of 553. For comparison, the human reference is of 100 tokens. The average latency of the responses from OpenRouter is of 1471 milliseconds, with a min of 288 milliseconds and a max of 6120 milliseconds. The cost to run the experiments was approximately of one USD, not considering Deepseek since it was offered free of charge.

Only in eight instances more than one PGP is invoked, and it never results in plans that reach home. In this subset of experiments the PGP is called two times on average, with a single outlier in which the PGP is called nine times. All subsequent tables and figures exclude instances where the PGP was invoked more than once.

The analysis is structured as follows: section 5.3.1 shows how the LLMs were prompted, section 5.3.2 shows some example responses and section 5.3.3 provides a detailed breakdown of the outcomes.

5.3.1 Example PGP Prompt

The explorer robot application has been tested with a prompt that defines content both for the system and user roles. Listing 5.6 shows the system prompt sent in all the PGP invocations and listing 5.7 shows the user prompt with both hints and remarks included.

```
# System Message
You are a Belief-Desire-Intention (BDI) agent that devises plans to pursue goals.

## Core Principles
- The more general the plan, the better
- Encode beliefs as first-order-logic (FOL) facts
- Encode goals as FOL terms
- Encode plans as triplets: (event, condition, operation)

## Plan Structure
Plans have the format (event, condition, operation) where:
- **event**: the goal to be pursued
- **condition**: FOL formula tested against current beliefs
- **operation**: list of activities to perform

## Event Types
Events must be prefixed with keywords:
- `achieve`: goals the agent should actively work towards (e.g., "achieve reach(home)")

## Operation Types
Operations must be prefixed with keywords:
- `execute`: primitive actions that directly interact with environment (e.g., `execute
    move(north)`)
- `achieve`: set new subgoal that triggers another plan (e.g., `achieve reach(rock)`)
- `add`: add new belief to belief base (e.g., `add visited(current_location)`)
- `remove`: remove existing belief (e.g., `remove obstacle(north)`)
- `update`: modify existing belief (e.g., `update position(X, Y)`)

## Output Format
Always format plans as:
```yaml
EVENT: achieve event to be pursued
CONDITIONS:
  - condition to be satisfied
  - other conditions to be satisfied
OPERATIONS:
  - [execute|achieve|add|remove|update] operation to be performed
  - [execute|achieve|add|remove|update] other operations to be performed
```

Separate multiple plans with `---`. Use `<none>` for empty conditions or operations.

Always format newly invented admissible goals and beliefs as:
```yaml
- goal: `my_goal(X)`
  purpose: natural language interpretation of my_goal for a general X
- belief: `my_belief(Y)`
  purpose: natural language interpretation of my_belief for a general Y
- ...
```

Constraints
- Use FOL syntax with no quantifiers
- Be as general and minimal as possible
- Use variables instead of constants where appropriate
- Reuse patterns across plans
- Cannot invent new admissible actions
- Must use all invented admissible goals/beliefs
- Cannot reference existing admissible goals/beliefs when inventing new ones
```

Listing 5.6: The system prompt used by the robot explorer application.

```
# User Message
Below is your internal state and the specific goal I need you to plan for.

## Agent's internal state
### Beliefs
#### Admissible beliefs
- obstacle(Direction): there is an obstacle to the `Direction`
- there_is(Object, Direction): there is an `Object` in the given `Direction`
- direction(Direction): `Direction` is a direction
- object(Object): `Object` is an object

#### Actual beliefs
- direction(north): north is a direction
- direction(south): south is a direction
- direction(east): east is a direction
- direction(west): west is a direction
- direction(north_east): north_east is a direction
- direction(north_west): north_west is a direction
- direction(south_east): south_east is a direction
- direction(south_west): south_west is a direction
- direction(here): here is a direction
- object(rock): rock is an object
- object(home): home is an object
- free(north): there is no obstacle to the north
- free(east): there is no obstacle to the east
- free(west): there is no obstacle to the west
- free(north_east): there is no obstacle to the north_east
- free(north_west): there is no obstacle to the north_west
- obstacle(south): there is an obstacle to the south
- obstacle(south_east): there is an obstacle to the south_east
- obstacle(south_west): there is an obstacle to the south_west

### Goals
#### Admissible goals
- achieve reach(Object): reach a situation where `Object` is in the position of the agent (i.e.
    there_is(`Object`, here))

#### Actual goals
None.

### Admissible actions
- fail(): makes the agent fail its current intention
- stop(): stops the agent
- pause(): pauses the agent
- sleep(Time): makes the agent sleep for `Time` milliseconds
- move(Direction): move in the given direction: `Direction`
- getDirectionToMove(Direction): provides a Direction free of obstacles where the agent can then
    move

## Expected outcome
Create plans to pursue the goal: achieve reach(home).
Output only the final set of plans with no alternatives or intermediate attempts.
End with an additional YAML block that contains a list of any new admissible goals and beliefs
    you invented, including their natural language interpretation.
```

Listing 5.7: The user prompt used by the robot explorer application.

5.3.2 Example PGP Responses

In listing 5.8, listing 5.9, listing 5.10 and listing 5.11 are shown example responses for each of the tested models. The responses are obtained by extracting the raw message content and are not preprocessed in any way.

````
```yaml
EVENT: achieve reach(home)
CONDITIONS:
  - there_is(home, here)
OPERATIONS:
  - <none>
```

---

```yaml
EVENT: achieve reach(home)
CONDITIONS:
  - there_is(home, Direction)
  - direction(Direction)
OPERATIONS:
  - execute getDirectionToMove(Direction)
  - execute move(Direction)
```

---

```yaml
EVENT: achieve reach(home)
CONDITIONS:
  - <none>
OPERATIONS:
  - execute getDirectionToMove(Direction)
  - execute move(Direction)
  - achieve reach(home)
```

```yaml
- <none>
```
````

Listing 5.8: Example response given by Claude Sonnet 4 (beautiful-volhard). This is one of the few successful PGP invocations which use a prompt with no hints or remarks.

5.3.3 Detailed Metrics' Analysis

This section examines the preliminary results obtained from the experimental evaluation, focusing on the specific metrics and their implications for PGP perfor-

````
```yaml
EVENT: achieve reach(home)
CONDITIONS:
  - there_is(home, here)
OPERATIONS:
  - execute stop()
---

EVENT: achieve reach(home)
CONDITIONS:
  - there_is(home, Direction)
  - free(Direction)
OPERATIONS:
  - execute move(Direction)
  - achieve reach(home)
---

EVENT: achieve reach(home)
CONDITIONS:
  - there_is(home, Direction)
  - obstacle(Direction)
OPERATIONS:
  - execute getDirectionToMove(NewDirection)
  - execute move(NewDirection)
  - achieve reach(home)
---

EVENT: achieve reach(home)
CONDITIONS:
  - <none>
OPERATIONS:
  - execute getDirectionToMove(Direction)
  - execute move(Direction)
  - achieve reach(home)
```

```yaml
- goal: `explore()`
  purpose: explore the environment to find the home object when its location is unknown
- belief: `visited(Location)`
  purpose: indicates that the agent has already visited this location
```
````

Listing 5.9: Example response given by Deepseek V3 Chat (stoic-rhodes).

````
```yaml
EVENT: achieve reach(X)
CONDITIONS:
  - object(X)
  - not there_is(X, here)
OPERATIONS:
  - achieve explore_for(X)
---
EVENT: achieve explore_for(X)
CONDITIONS:
  - object(X)
  - not there_is(X, D)
OPERATIONS:
  - execute getDirectionToMove(D)
  - execute move(D)
  - achieve explore_for(X)
---
EVENT: achieve explore_for(X)
CONDITIONS:
  - object(X)
  - there_is(X, D)
OPERATIONS:
  - achieve approach(X, D)
---
EVENT: achieve approach(X, D)
CONDITIONS:
  - object(X)
  - direction(D)
  - there_is(X, D)
  - not there_is(X, here)
OPERATIONS:
  - execute move(D)
  - achieve reach(X)
---
EVENT: achieve reach(X)
CONDITIONS:
  - object(X)
  - there_is(X, here)
OPERATIONS:
  - <none>
```
```yaml
- goal: `explore_for(X)`
  purpose: actively search for the location of object X when its location is unknown
- goal: `approach(X, D)`
  purpose: move towards object X in direction D when X is adjacent but not yet reached
```
````

Listing 5.10: Example response given by GPT 4.1 (stupefied-lewin).

````
```yaml
EVENT: achieve reach(Object)
CONDITIONS:
  - there_is(Object, here)
OPERATIONS:
  - <none>
---
EVENT: achieve reach(Object)
CONDITIONS:
  - NOT there_is(Object, here)
  - NOT there_is(Object, Direction)
OPERATIONS:
  - achieve explore()
  - achieve reach(Object)
---
EVENT: achieve reach(Object)
CONDITIONS:
  - NOT there_is(Object, here)
  - there_is(Object, Direction)
  - direction(Direction)
OPERATIONS:
  - achieve move_towards(Direction)
  - achieve reach(Object)
---
EVENT: achieve move_towards(Direction)
CONDITIONS:
  - direction(Direction)
  - free(Direction)
OPERATIONS:
  - execute move(Direction)
---
EVENT: achieve move_towards(Direction)
CONDITIONS:
  - direction(Direction)
  - obstacle(Direction)
OPERATIONS:
  - execute fail()
---
EVENT: achieve explore()
CONDITIONS:
  - <none>
OPERATIONS:
  - execute getDirectionToMove(Direction)
  - achieve move_towards(Direction)
```

```yaml
- goal: `achieve explore()`
  purpose: find new areas to search for objects
- goal: `achieve move_towards(Direction)`
  purpose: move in a specific direction if possible
```
````

Listing 5.11: Example response by Gemini 2.5 Flash (frosty-kare). This is the response that got the highest score by the LLM judge, which coincidentally was Gemini 2.5 Flash itself, and was one of the few answers the model provided that were successful.

mance. While these findings provide valuable insights into model behavior and structural plan characteristics, their broader applicability remains limited due to the constrained experimental scope, the limited set of models examined and the limited number of samples. Additional experimentation across diverse model architectures and domains will be necessary to establish the generalizability of these observations and to develop a more robust understanding of the factors influencing the PGP performance.

Grouping by Prompt and Model

Table 5.2, fig. 5.3 and fig. 5.5 present an evaluation of the defined metrics for each of the chosen models under three different prompting conditions. The performance varies significantly according to the type of prompt employed, indicating sensitivity to prompt engineering. In particular the best performance in terms of **TSR** is achieved when LLMs are given the most rich context, while they seem to struggle to generate a successful answer at all if no kind of documentation is given. This suggests that, when Generative BDI agents are used, the role of the programmer in providing a natural language interpretation of the domain concepts expressed through beliefs, goals and actions is critical.

Grouping by Temperature and Model

Table 5.3, fig. 5.4 and fig. 5.6 show how the value of the metrics change as the temperature varies. Each model exhibits a varying sensitivity to the temperature, responding optimally under different configurations. Claude has a low **TSR** when operating at low temperature settings (0.1). Both Claude and Gemini achieve better results when configured with moderate temperature values (0.5), suggesting they benefit from an approach that balances determinism with creative outputs. Deepseek shows robust performance across both low and moderate temperature settings, which might imply greater stability across different parameter ranges. In contrast, GPT performs most effectively when configured with a low temperature. These findings suggest that optimal temperature selection is model-dependent, with some architectures requiring more randomness to achieve peak performance while others excel under more constrained conditions.

Grouping by Temperature, Prompt Type and Model

Table 5.4 provides an overview of the best performing combinations of parameters considering both the temperature and the prompt type. GPT 4.1 and Deepseek report the highest **TSR** score, both achieving a perfect result with a temperature of 0.1 and by providing both hints and remarks in the prompt. However, while Deepseek maintains perfect performance even at temperature 0.5, GPT 4.1 shows significant degradation at this setting, dropping to a **TSR** of 70. When prompts lack remarks, both top-performing models experience notable drops of their score, though they still maintain a relatively high **TSR** of around 90. This suggests that contextual information plays a crucial role in their effectiveness.

Claude stands out as the only model capable of achieving a success when no hints are provided, albeit with modest **TSR** values of around 10, which might imply that is the most robust to information scarcity. It's interesting to note that only with higher temperature values (0,5 and 0,9) the model achieves those successes, suggesting that when there is too little information, more "creative exploration" helps in finding a valid solution to ambiguous or under-specified problems. This is just a speculation since more experimental runs would be needed to assess whether this is a random fluctuation or a pattern, given the very low **TSR**. Nevertheless, this observation might inform future iterations of the PGP, where dynamic temperature adjustment could emulate divergent and convergent thinking patterns. A higher value might be used during the initial exploration or when the PGP gets stuck, while a lower one can be used to focus efforts on refining viable solutions.

Gemini is the lowest performing model, showing instability at both temperature extremes and achieving only occasional successes when working with the most information-rich variant of the prompt.

Results considering only successful PGP invocations

The results in table 5.5 consider only successful PGP invocations. GPT 4.1 and Gemini demonstrate the highest generalization count (**GC** scores of 4.21 and 4.67 respectively), indicating their enhanced ability to develop flexible, variable-based plans instead of rigid, hard-coded approaches. Despite Gemini's strong generalization performance, it struggles to convert this advantage into reliable task completion (see table 5.4), unlike GPT 4.1 which maintains consistent success rates. This pattern suggests that plan generality doesn't necessarily guarantee execution effectiveness. The reverse relationship also appears true—execution success doesn't ensure plan reusability. Deepseek exemplifies this disconnect, achieving high **TSR** while reporting the lowest generalization scores. This indicates that models can excel at completing specific tasks without developing transferable solution strategies. Furthermore, both GPT and Gemini exhibit high **GAT** scores, whereas Deepseek is the only model that consistently approaches human-level performance (**GAT** 33), with a score of 37. This observation suggests that highly generalized solutions occasionally sacrifice efficiency compared to task-specific approaches.

For what concerns the plan complexity metrics, Claude, and in greater measure Deepseek, exhibit a distinct pattern: they generate complex plan bodies (resp. **PBC** of 1.67 and 2.22) while maintaining moderately complex plan contexts (resp. **CC** of 1.41 and 1.33). This complexity likely stems from their low generalization capability (resp. **GC** of 0.57 and 0.08), as avoiding variables naturally results in more verbose and repetitive code structures where similar operations are explicitly written out rather than abstracted into reusable components. In contrast, GPT 4.1 demonstrates the inverse pattern, producing the most complex contexts (**CC** 1.96) while generating less complex bodies (**PBC** 1.53). These results seem to align with its higher generalization performance (**GC** 4.21), since the model handles diverse contingencies within the context and generates short bodies through the use of variables, representing the opposite approach to the strategy of Deepseek and Claude. Gemini showed the highest generalization capability, with a **GC** of 4.67.

Across all models, redundancy rates remained low, indicating that generated plans rarely subsumed one another. This suggests that while models may vary in their generalization and complexity strategies, they consistently produce distinct solutions rather than generating overlapping or redundant plans.

Regarding goal and belief invention, Deepseek shows a marked tendency to invent new beliefs, whereas GPT leans more toward inventing goals. Claude has the lowest combined score, indicating the least amount of invention overall. That said, the overall **NGC** and **NBC** scores are quite low, suggesting that the models generally understood that no new goals or beliefs were necessary for the given domain.

Considering the semantic alignment metrics, the **BSA** is relatively high, especially for Deepseek and Gemini. This is often due to the models introducing new beliefs that are ultimately unused. In contrast, the **GSA** is quite low. This suggests that when new goals were invented, especially by GPT-4.1 and Gemini, they were effectively integrated and utilized. Overall, the models appear to have followed the constraint of not inventing goals or beliefs that already existed in the domain, showing good semantic alignment in relation to the human reference.

Plan-Reference Alignment Score

The data previously presented in the tables reveals substantial divergence between the generated plans and the baseline reference. This divergence manifests both in the structure and complexity of the generated plans: the plan counts are consistently higher and both the context and body complexity are higher compared to the baseline.

The **PRAS** evaluation scores in fig. 5.2 seems to reinforce this structural misalignment, due to the uniformly low scores across all models. Notably, Gemini, the model that obtained the highest **PRAS** score, is also the one with the lowest **TSR**.

Model	**PT**	**PC**	**CC**	**PBC**	**GC**	**RR**	**NGC**	**NBC**	**GSA**	**BSA**	**TSR**
Claude	NH	5.17	1.88	1.52	3.86	0.00	2.21	1.21	0.41	1.76	6.90
Claude	H	5.14	1.82	1.27	2.00	0.00	1.90	0.07	0.00	2.00	0.00
Claude	HR	3.41	1.40	1.61	0.03	0.00	0.03	0.00	0.38	1.72	65.52
Deepseek	NH	3.64	1.86	2.01	0.50	0.00	1.00	1.25	0.25	2.29	0.00
Deepseek	H	3.54	1.85	1.97	0.00	0.00	0.25	1.21	0.25	3.36	50.00
Deepseek	HR	3.57	1.40	2.08	0.30	0.00	0.37	1.03	0.47	3.47	80.00
GPT 4.1	NH	3.53	2.65	1.91	3.53	0.00	1.20	1.03	0.37	1.87	0.00
GPT 4.1	H	4.23	2.06	1.71	4.23	0.03	1.70	0.90	0.13	1.13	63.33
GPT 4.1	HR	4.47	2.00	1.36	4.37	0.00	1.57	0.07	0.00	0.57	80.00
Gemini	NH	3.14	1.63	1.49	3.14	0.03	0.17	0.31	0.31	2.17	0.00
Gemini	H	4.82	1.48	1.53	4.79	0.64	0.71	1.79	0.54	2.86	0.00
Gemini	HR	4.04	1.13	1.13	3.96	0.00	0.86	0.18	0.18	1.75	10.71

Table 5.2: Results grouped by model and prompt type, with metrics averaged. **PT** stands for the parameter “Prompt Type”. **NH** stands for “NoHints”, **H** for “WithHints”, and **HR** for “WithHintsAndRemarks”.

Model	T	PC	CC	PBC	GC	RR	NGC	NBC	GSA	BSA	TSR
Claude	0.1	4.80	1.74	1.37	2.00	0.00	1.43	0.40	0.43	1.93	16.67
Claude	0.5	4.44	1.69	1.51	2.04	0.00	1.41	0.33	0.11	1.56	29.63
Claude	0.9	4.47	1.67	1.52	1.87	0.00	1.30	0.53	0.23	1.97	26.67
Deepseek	0.1	3.52	1.60	2.15	0.14	0.00	0.34	1.14	0.10	3.48	62.07
Deepseek	0.5	3.46	1.60	2.07	0.29	0.00	0.57	1.25	0.39	3.14	46.43
Deepseek	0.9	3.76	1.89	1.86	0.38	0.00	0.69	1.10	0.48	2.52	24.14
GPT 4.1	0.1	3.77	2.25	1.70	3.77	0.00	1.50	0.50	0.20	0.93	63.33
GPT 4.1	0.5	4.17	2.15	1.61	4.13	0.00	1.53	0.50	0.13	1.10	43.33
GPT 4.1	0.9	4.30	2.32	1.66	4.23	0.03	1.43	1.00	0.17	1.53	36.67
Gemini	0.1	4.10	1.35	1.20	4.07	0.40	0.67	1.33	0.33	3.00	0.00
Gemini	0.5	4.21	1.46	1.47	4.14	0.21	0.52	0.41	0.24	1.76	10.34
Gemini	0.9	3.62	1.44	1.50	3.62	0.04	0.54	0.46	0.46	1.96	0.00

Table 5.3: Results grouped by model and temperature, with metrics averaged. **T** stands for the parameter "Temperature".

Model	Prompt Type	Temperature	TSR
GPT 4.1	Hints and Remarks	0.1	100.00
Deepseek	Hints and Remarks	0.1	100.00
Deepseek	Hints and Remarks	0.5	100.00
GPT 4.1	Only Hints	0.1	90.00
Deepseek	Only Hints	0.1	88.89
Claude	Hints and Remarks	0.5	77.78
GPT 4.1	Hints and Remarks	0.9	70.00
GPT 4.1	Hints and Remarks	0.5	70.00
Claude	Hints and Remarks	0.9	70.00
GPT 4.1	Only Hints	0.5	60.00
Claude	Hints and Remarks	0.1	50.00
GPT 4.1	Only Hints	0.9	40.00
Deepseek	Hints and Remarks	0.9	40.00
Gemini	Hints and Remarks	0.5	33.33
Deepseek	Only Hints	0.9	33.33
Deepseek	Only Hints	0.5	30.00
Claude	No Hints	0.5	11.11
Claude	No Hints	0.9	10.00

Table 5.4: Results grouped by prompt type, temperature and model, sorted by task success rate and with metrics averaged. Entries with a **TSR** of zero were omitted.

Model	PC	CC	PBC	GC	RR	NGC	NBC	GSA	BSA	GAT
Deepseek	3.50	1.41	2.22	0.08	0.00	0.21	1.11	0.29	3.79	37.71
Claude	3.38	1.33	1.67	0.57	0.00	0.38	0.29	0.05	1.10	247.52
Gemini	5.33	1.17	1.31	4.67	0.00	1.33	0.33	0.33	2.67	506.33
GPT 4.1	4.21	1.96	1.53	4.21	0.00	1.60	0.42	0.02	0.70	1033.16
Optimal	3.00	1.00	1.30	3.00	0.00	0.00	0.00	0.00	0.00	33.00

Table 5.5: Results grouped by model and averaged across all metrics, considering only successful PGP executions.

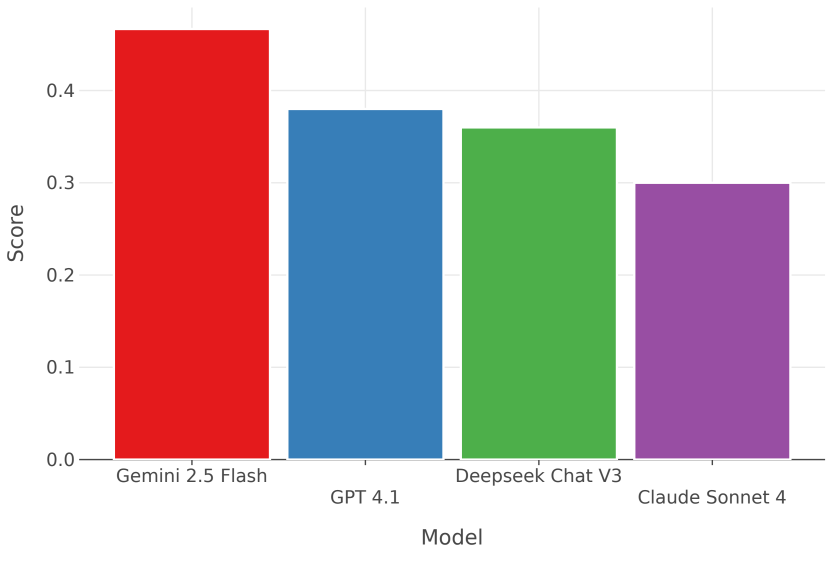


Figure 5.2: The average **PRAS** score for each model was calculated by sampling up to five PGP results from the set of successful PGP invocations.

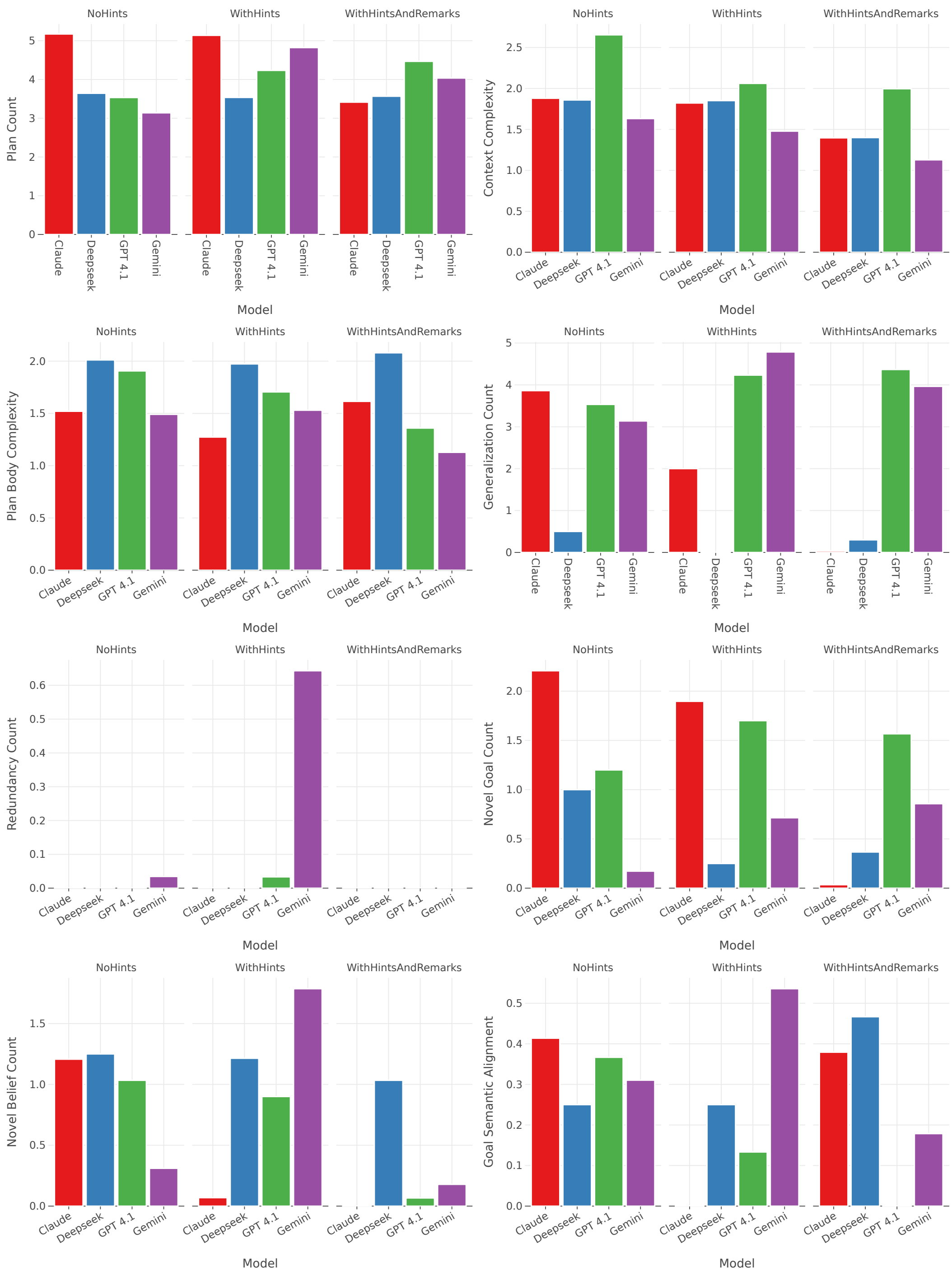


Figure 5.3: Bar plots showing **PC**, **CC**, **PBC**, **GC**, **RR**, **NGC**, **NBC** and **GSA** scores for each model, with a facet for each prompt type tested.

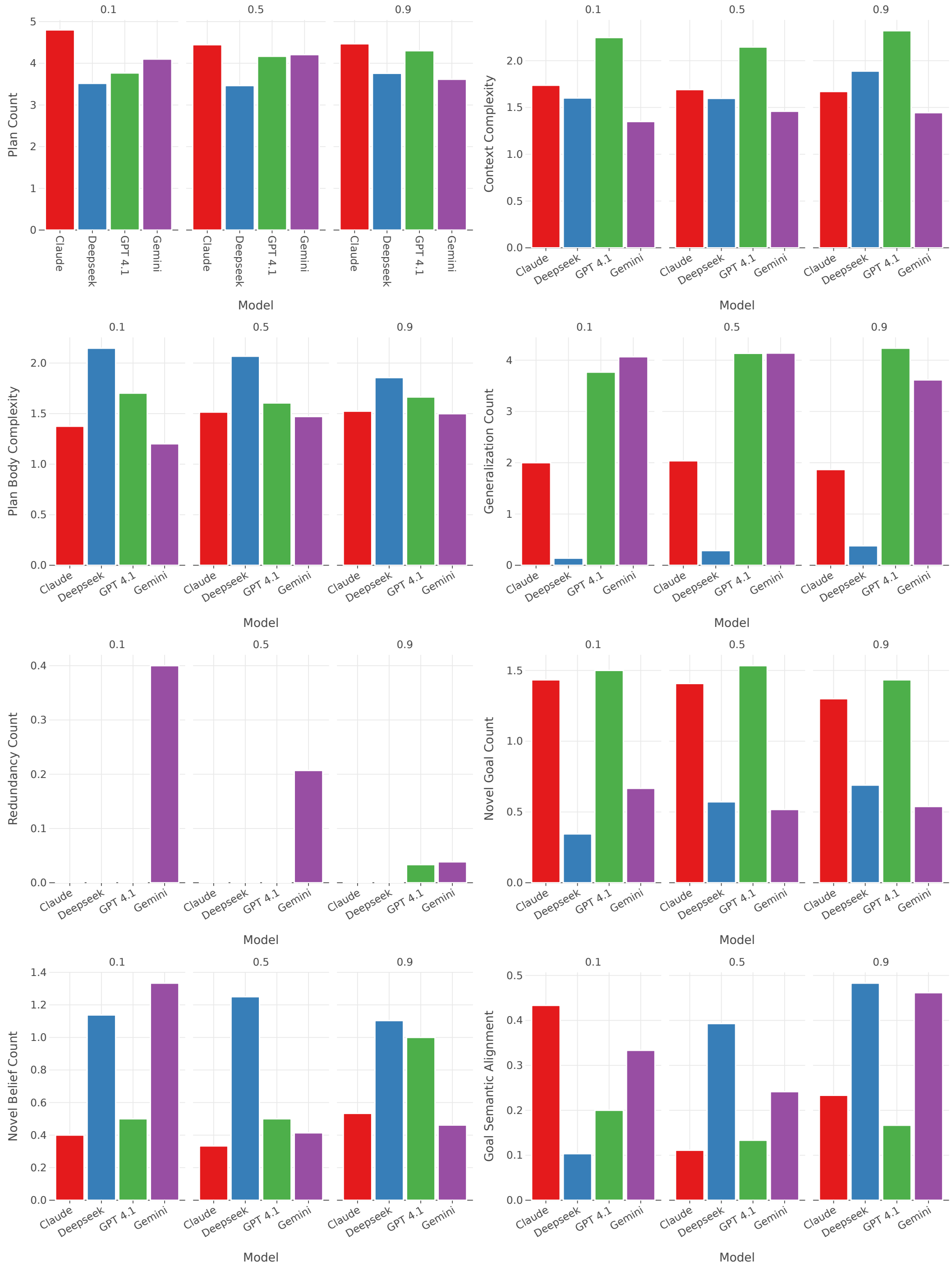


Figure 5.4: Bar plots showing **PC**, **CC**, **PBC**, **GC**, **RR**, **NGC**, **NBC** and **GSA** scores for each model, with a facet for each value of temperature tested.

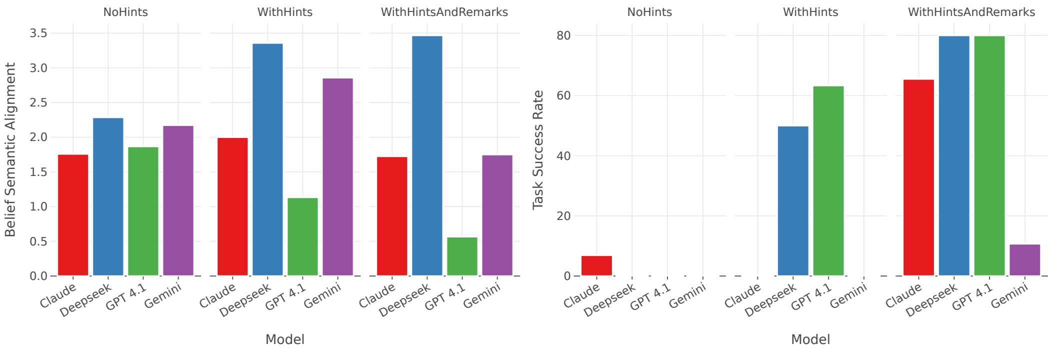


Figure 5.5: **BSA** and **TSR** scores for each model, with a facet for each prompt type tested.

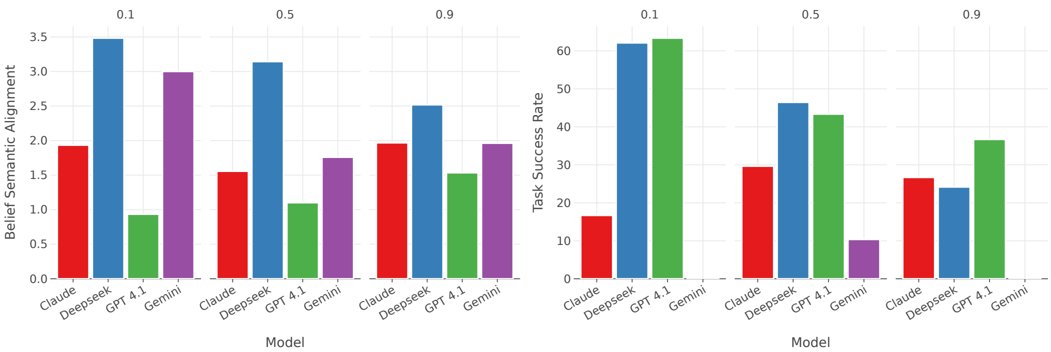


Figure 5.6: **BSA** and **TSR** scores for each model, with a facet for each value of temperature tested.

Chapter 6

Conclusion

This thesis focuses on the integration of GenAI into the AgentSpeak(L) agent architecture to generate new plans at runtime, based on the agent's current knowledge. The goal is to assess if and how LLMs can generate plans for BDI agents, potentially reducing reliance on human programmers or first-principle planners.

Unlike approaches that treat agents as generative systems themselves, this work preserves the strengths of the BDI model—its theoretical foundation, programming paradigms, and controllability—while enhancing it with generative and natural language processing (NLP) capabilities for plan generation. The approach encapsulates the LLM within a PGP functionality, responsible for generating plans reactively or on-demand, ensuring compatibility with the BDI architecture. This thesis defines the PGP interface, proposes implementation guidelines, and evaluates a prototype implementation to validate the approach.

Addressing item **(RQ1)**, the PGP is defined to use structured prompts detailing current and generally *admissible* goals and beliefs, available actions and plans, and the operational semantics of the BDI architecture.

For item **(RQ2)**, structured prompts paired with parsable outputs ensure reliable integration of generated plans into the agent's library, balancing expressiveness with formal rigor. As experimental results confirm, providing additional semantics to the LLM via natural language descriptions of the agent's goals, beliefs, and actions has a critical impact on the effectiveness of the LLM-based PGP, along with the application of prompt engineering best practices and appropriate sampling parameters.

Consequently, answering item **(RQ3)**, automatic plan generation shifts the agent specification from exhaustive plan encoding to providing the basic (procedural) knowledge necessary for the problem domain, potentially leaving to the LLM the task of handling corner cases and unexpected situations. The agent operation stays the same of a classic BDI agent, with the possibility of the PGP to be triggered explicitly (on-demand) or implicitly (reactively) to handle goals with no matching plans.

For item **(RQ4)**, the methodology outlined in this work shows promise for the generation of reusable, general plans involving variables, despite variability in LLM performance.

In conclusion, this thesis bridges traditional BDI architectures and emerging GenAI technologies, laying the groundwork for more autonomous and explainable cognitive agents and opening several promising research directions.

6.1 Future Work

Future work includes exploring runtime validation and verification mechanisms to address hallucinations or inaccuracies in LLM-generated constructs. Several other research directions might be explored, including plan repair and refinement mechanisms (§ 6.1.1), experimentation with different structured output formats (§ 6.1.2), enhanced modularization of BDI agent cycles (§ 6.1.3), integration with Model Context Protocol (MCP) (§ 6.1.4), finetuning (§ 6.1.5) and use of artifacts (§ 6.1.6).

Additionally, studying prompt robustness under varying conditions and developer inaccuracies could enhance the methodology's practicality. Ablation studies may clarify factors influencing prompt effectiveness **(RQ1)** and knowledge transfer **(RQ2)**. Testing scenarios with concurrent goals, dynamic environments, and partial observability will validate scalability and generalization **(RQ4)**.

6.1.1 Plan repair and refinement mechanisms

Mechanisms such as plan repair, failure learning, and adaptive refinement can enhance agent autonomy and long-term performance. One effective strategy involves incorporating feedback from external entities. This has been successfully employed to back-prompt LLMs for improved plan generation.

As an example Voyager uses the feedback from Minecraft to provide the LLM with up-to-date information on how the environment changed in response to its chosen actions and whether syntax errors led to failure in executing plans [42]. Analogously, Kambhampati et al. [35] demonstrate that within the LLM-Modulo framework, incorporating external verifiers or critics can significantly enhance the planning capabilities of language models. Their view is encapsulated in the assertion: "LLMs cannot plan themselves but can play a variety of constructive roles in solving planning tasks—especially as approximate knowledge sources and candidate plan generators in so-called LLM-Modulo Frameworks, where they are used in conjunction with external sound model-based verifiers" [35]. This approach highlights the use of LLMs as generators of candidate solutions, with their correctness checked through independent verification mechanisms. In this context, BDI engines are particularly well-suited to serve as verifiers, offering detailed feedback on the execution of LLM-generated plans.

6.1.2 Structured Output Formats

YAML was chosen as the structured format for plan generation due to the ease with which it can be parsed and its readability, which is greater than the one of XML or JSON. Additionally, the format's prevalence in training data, for example through configuration files, was hypothesized to result in more reliable generation compared to specialized agent programming languages like AgentSpeak(L).

Future work with controlled natural languages [63] might be a promising direction for improving plan generation while addressing current limitations. Natural language alternatives could potentially offer higher generation quality due to training data alignment—since the vast majority of LLM training corpora consists of natural language text from books, articles, and web content rather than structured formats—potentially increasing the quality of the generated plans. This alignment also facilitates easier human review and modification, and better domain expertise integration through natural descriptions, though at the cost of more complex parsing.

Prompting with grammatical patterns could significantly improve parsing reliability, for example, by using standardized phrasings like "when attempting to achieve [goal] given [conditions], execute [actions]" that map deterministically to BDI constructs. Template-based controlled generation represents a complementary avenue, where LLMs fill structured natural language templates that encode BDI semantics while maintaining readability, potentially combined with constraint-based generation that enforces grammatical rules as part of the generation process. For what concerns this work, the choice of using YAML prioritizes the integration with the BDI interpreter over the theoretical advantages of native agent programming language output or the contextual richness of natural language planning descriptions.

6.1.3 Improved Software Modularity

Plan Generation. Despite all the LM-related logic being implemented in a separate module, the PGP is still tightly coupled with the BDI engine. The sense-reason-act cycle of a BDI agent could be further decomposed into pluggable components to support dynamic composition and substitution of agent functionalities through external modules. This approach would enable users to define in a declarative way how an agent's cognitive cycle is composed as part of its specification, e.g. what component to use for plan selection or for running intentions. Such a modular architecture would make it easier to separate all plan generation functions from the BDI engine. Plan generation could be integrated by adjusting the deliberation phase to handle cases where no plan is available and enhancing the action phase to manage `GeneratePlan` and `TrackGoalExecution` goals.

Logging System. The current logging implementation is tightly coupled with the BDI engine. An event-driven architecture can be implemented, by leveraging Kotlin Flows, to define a separate logging module that registers as a listener for events emitted by the BDI engine.

6.1.4 Model Context Protocol Integration

A promising direction for enhancing LLM-augmented BDI agent systems involves leveraging MCP, an open standard that enables to build connections between data sources and GenAI-powered applications. Unlike the current implementation of the PGP, which requires custom interfaces for each external tool or data source the BDI agent accesses—such as the logic to retrieve the agent's context—MCP provides a standardized, reusable mechanism to integrate diverse components. MCP facilitates the construction of flexible, composable pipelines that orchestrate retrieval, transformation, and reasoning components. This enables more sophisticated plan-generation workflows, where the LLM can draw from heterogeneous sources such as knowledge graphs, databases, reasoning tools, or other APIs.

A potential application of this approach is to integrate with a MCP server that performs Retrieval-Augmented Generation (RAG). This can help reduce the size of the LLM prompt, which in turn improves inference times and reduces token consumption. Instead of passing the entire agent context to the LLM, the RAG pipeline would extract only the most relevant subset, based for example on the current goal or plan-in-progress.

Furthermore, the same MCP-based infrastructure could enable the LLM to reason over execution traces of other agents—e.g., to detect cooperative or competitive behaviors and act accordingly. Specialized MCP connectors could expose the agent's belief base, intention stack, and historical plans as structured knowledge.

This architecture would not only improve the performance and adaptability of LLM-enhanced BDI agents but also enable better reuse, composability, and scalability across a range of agent configurations and environments.

6.1.5 Finetuning Small Language Models

An advantage of the proposed PGP is that it works with pretrained LLMs without any kind of finetuning. That said, it is worth considering the opportunity of finetuning small language models (SLMs), especially since with current state-of-the-art models results seem promising.

Domain-specific finetuning could significantly improve plan generation quality for specialized applications. Training on curated datasets of high-quality plans in specific domains can enhance the model's understanding of domain constraints, best practices, and common failure modes.

Using SLMs instead of large-scale LLMs also offers notable advantages in terms of efficiency and speed since SLMs are faster and less resource-intensive.

6.1.6 Artifacts as Tools

Future developments could explore the incorporation of artifacts—and more specifically, cognitional artifacts—as a means to further structure the interaction between generative BDI agents and the environment.

Artifacts in the A&A [64] are computational entities designed to be used by agents, exposing functionality through a well-defined *usage interface*, and optionally enriched with *function descriptions* and *operating instructions*. These features make artifacts naturally compatible with LLMs, as they embody a set of properties increasingly characteristic of LLM-native software development.

Artifacts provide *introspectability* by exposing their purpose and usage in a form that can be reasoned about symbolically or through language. They offer *explainability* through function and operation descriptions that can be explicitly described in natural language, making them amenable to LLM understanding and debugging. Additionally, artifacts provide *modularity* by offering encapsulated, reusable functionality that could be invoked or composed by LLM-generated plans.

Recent trends in GenAI highlight the shift toward language-accessible software, where components are made self-descriptive and navigable by LLMs. Cognitional artifacts—by design—already promote this pattern, and could serve as interfaces or tools that LLMs can select, understand, and use during plan generation.

As such, a promising future direction is the development of artifact-based infrastructures in which LLMs reason over available artifacts via their function descriptions, generate plans informed by artifact operating instructions, and execute them through artifact usage interfaces.

This would also support a more declarative, open-ended, and cognitively rich model of agent behavior, aligning with the *Agens Faber* vision, where agent intelligence includes the capacity to understand, use, and even construct new artifacts.

Bibliography

[1] Anand S. Rao and Michael P. Georgeff. “BDI Agents: From Theory to Practice”. In: *1st International Conference on Multi Agent Systems (ICMAS 1995)*. Ed. by Victor R. Lesser and Les Gasser. San Francisco, CA, USA: The MIT Press, Dec. 1995, pp. 312–319. ISBN: 0-262-62102-9. URL: `https://www.aaai.org/Papers/ICMAS/1995/ICMAS95-042.pdf` (cit. on pp. 3, 5, 6).

[2] Anand S. Rao. “AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language”. In: *Agents Breaking Away*. Ed. by Walter Van de Velde and John W. Perram. Vol. 1038. Lecture Notes in Computer Science. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’96). Springer, 1996, pp. 42–55. ISBN: 978-3-540-60852-3. DOI: `10.1007/BFb0031845` (cit. on p. 3).

[3] Lavindra de Silva, Sebastian Sardiña, and Lin Padgham. “First principles planning in BDI systems”. In: *8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*. Ed. by Carles Sierra et al. IFAAMAS, 2009, pp. 1105–1112. URL: `https://dl.acm.org/citation.cfm?id=1558167` (cit. on p. 3).

[4] Felipe Meneguzzi and Lavindra de Silva. “Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning”. In: *Knowl. Eng. Rev.* 30.1 (2015), pp. 1–44. DOI: `10.1017/S0269888913000337`. URL: `https://doi.org/10.1017/S0269888913000337` (cit. on pp. 3, 10).

[5] Xu Huang et al. “Understanding the planning of LLM agents: A survey”. In: *CoRR* abs/2402.02716 (2024). URL: `https://doi.org/10.48550/arXiv.2402.02716` (cit. on p. 4).

[6] Giovanni Ciatto et al. "Large language models as oracles for instantiating ontologies with domain-specific knowledge". In: *Knowledge-Based Systems* 310 (15 2 2025), 112940:1–22. ISSN: 0950-7051. DOI: `10.1016/j.knosys.2024.112940`. URL: `https://www.sciencedirect.com/science/article/pii/S0950705124015740` (cit. on p. 4).

[7] Victor Aregbede et al. "Affordance-Based Goal Imagination for Embodied AI Agents". In: *2024 IEEE International Conference on Development and Learning (ICDL)*. 2024, pp. 1–6. DOI: `10.1109/ICDL61372.2024.10644764` (cit. on p. 4).

[8] Martina Baiardi et al. "Blending BDI agents with object-oriented and functional programming with JaKtA". In: *Springer Nature Computer Science (SNCS)* 5.1003 (2024). DOI: `10.1007/s42979-024-03244-y` (cit. on pp. 4, 9).

[9] Michael Bratman et al. *Intention, plans, and practical reason*. Vol. 10. Cambridge, MA: Harvard University Press, 1987 (cit. on p. 5).

[10] Anand S. Rao and Michael P. Georgeff. "Decision procedures for BDI logics". In: *Journal of Logic and Computation* 8.3 (June 1998), pp. 293–342. ISSN: 0955-792X. DOI: `10.1093/logcom/8.3.293`. URL: `http://logcom.oxfordjournals.org/cgi/content/long/8/3/293` (cit. on p. 5).

[11] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Oxford, England: John Wiley & Sons, 2007 (cit. on pp. 5, 18).

[12] Jomi Fred Hübner and Rafael H. Bordini. "Agent-Based Simulation Using BDI Programming in Jason". In: *Multi-Agent Systems - Simulation and Applications*. 2009, pp. 451–476. DOI: `10.1201/9781420070248.CH15` (cit. on p. 5).

[13] Carole Adam and Benoit Gaudou. "BDI agents in social simulations: a survey". In: *The Knowledge Engineering Review* 31.3 (2016), pp. 207–238 (cit. on p. 5).

[14] F.F. Ingrand, M.P. Georgeff, and A.S. Rao. “An architecture for real-time reasoning and system control”. In: *IEEE Expert* 7.6 (1992), pp. 34–44. DOI: 10.1109/64.180407 (cit. on p. 6).

[15] Mark d’Inverno et al. “The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System”. In: *Auton. Agents Multi Agent Syst.* 9.1-2 (2004), pp. 5–53. DOI: 10.1023/B:AGNT.0000019688.11109.19 (cit. on p. 6).

[16] Lavindra De Silva, Felipe Meneguzzi, and Brian Logan. “BDI Agent Architectures: A Survey”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. Yokohama, Japan: International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 4914–4921. ISBN: 978-0-9992411-6-5. DOI: 10.24963/ijcai.2020/684. (Visited on 06/10/2025) (cit. on p. 6).

[17] Michael Winikoff et al. “Declarative & Procedural Goals in Intelligent Agent Systems”. In: *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*. Ed. by Dieter Fensel et al. Morgan Kaufmann, 2002, pp. 470–481 (cit. on pp. 6, 7, 10).

[18] Jomi Fred Hübner, Rafael H. Bordini, and Michael J. Wooldridge. “Programming Declarative Goals Using Plan Patterns”. In: *Declarative Agent Languages and Technologies IV, 4th International Workshop, DALT 2006, Hakodate, Japan, May 8, 2006, Selected, Revised and Invited Papers*. Ed. by Matteo Baldoni and Ulle Endriss. Vol. 4327. Lecture Notes in Computer Science. Springer, 2006, pp. 123–140. DOI: 10.1007/11961536_9. URL: https://doi.org/10.1007/11961536%5C_9 (cit. on p. 7).

[19] Koen V. Hindriks et al. “Agent Programming with Declarative Goals”. In: *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000, Boston, MA, USA, July 7-9, 2000, Proceedings*. Ed. by Cristiano Castelfranchi and Yves Lespérance. Vol. 1986. Lecture Notes in Computer Science. Springer, 2000, pp. 228–243. DOI: 10.1007/3-540-44631-1_16. URL: https://doi.org/10.1007/3-540-44631-1%5C_16 (cit. on p. 7).

[20] *Programming Multi-Agent Systems in AgentSpeak Using Jason*. 1st ed. John Wiley & Sons, Ltd, 2007. DOI: 10.1002/9780470061848. URL: https://onlinelibrary.wiley.com/doi/10.1002/9780470061848 (visited on 07/02/2025) (cit. on p. 9).

[21] Richard Fikes and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". In: *Artif. Intell.* 2.3/4 (1971), pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5. URL: https://doi.org/10.1016/0004-3702(71)90010-5 (cit. on p. 10).

[22] Ilche Georgievski and Marco Aiello. "HTN planning: Overview, comparison, and beyond". In: *Artif. Intell.* 222 (2015), pp. 124–156. DOI: 10.1016/J.ARTINT.2015.02.002. URL: https://doi.org/10.1016/j.artint.2015.02.002 (cit. on p. 10).

[23] Felipe Rech Meneguzzi and Michael Luck. "Leveraging New Plans in AgentSpeak(PL)". In: *Declarative Agent Languages and Technologies VI, 6th International Workshop, DALT 2008, Estoril, Portugal, May 12, 2008, Revised Selected and Invited Papers*. Ed. by Matteo Baldoni et al. Vol. 5397. Lecture Notes in Computer Science. Springer, 2008, pp. 111–127. DOI: 10.1007/978-3-540-93920-7_8. URL: https://doi.org/10.1007/978-3-540-93920-7%5C_8 (cit. on p. 10).

[24] Lavindra de Silva, Sebastian Sardiña, and Lin Padgham. "First principles planning in BDI systems". In: *8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*. Ed. by Carles Sierra et al. IFAAMAS, 2009, pp. 1105–1112. URL: https://dl.acm.org/citation.cfm?id=1558167 (cit. on p. 10).

[25] Rafael C. Cardoso, Louise A. Dennis, and Michael Fisher. "Plan Library Reconfigurability in BDI Agents". In: *Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13-14, 2019, Revised Selected Papers*. Ed. by Louise A. Dennis, Rafael H. Bordini, and Yves Lespérance. Vol. 12058. Lecture Notes in Computer Science. Springer, 2019, pp. 195–212. DOI: 10.1007/978-3-030-51417-4_10. URL: https://doi.org/10.1007/978-3-030-51417-4%5C_10 (cit. on p. 11).

[26] Jie Huang and Kevin Chen-Chuan Chang. "Towards Reasoning in Large Language Models: A Survey". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 1049–1065. DOI: `10.18653/v1/2023.findings-acl.67`. (Visited on 06/10/2025) (cit. on p. 11).

[27] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. Jan. 2023. arXiv: `2201.11903 [cs]`. (Visited on 10/26/2024) (cit. on p. 11).

[28] Takeshi Kojima et al. *Large Language Models Are Zero-Shot Reasoners*. Jan. 29, 2023. DOI: `10.48550/arXiv.2205.11916`. arXiv: `2205.11916 [cs]`. URL: `http://arxiv.org/abs/2205.11916` (visited on 06/10/2025). Pre-published (cit. on p. 11).

[29] R. Thomas McCoy et al. *Embers of Autoregression: Understanding Large Language Models Through the Problem They Are Trained to Solve*. Sept. 24, 2023. DOI: `10.48550/arXiv.2309.13638`. arXiv: `2309.13638 [cs]`. URL: `http://arxiv.org/abs/2309.13638` (visited on 12/16/2024). Pre-published (cit. on p. 11).

[30] R. Thomas McCoy et al. *When a Language Model Is Optimized for Reasoning, Does It Still Show Embers of Autoregression? An Analysis of OpenAI O1*. Oct. 4, 2024. DOI: `10.48550/arXiv.2410.01792`. arXiv: `2410.01792 [cs]`. URL: `http://arxiv.org/abs/2410.01792` (visited on 12/16/2024). Pre-published (cit. on p. 11).

[31] Zhaofeng Wu et al. *Reasoning or Reciting? Exploring the Capabilities and Limitations of Language Models Through Counterfactual Tasks*. Mar. 28, 2024. DOI: `10.48550/arXiv.2307.02477`. arXiv: `2307.02477 [cs]`. URL: `http://arxiv.org/abs/2307.02477` (visited on 06/10/2025). Pre-published (cit. on p. 11).

[32] Parshin Shojaee et al. "The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity". In: () (cit. on p. 11).

[33] John K. Slaney and Sylvie Thiébaux. “Blocks World revisited”. In: *Artif. Intell.* 125.1-2 (2001), pp. 119–153. DOI: 10.1016/S0004-3702(00)00079-5. URL: https://doi.org/10.1016/S0004-3702(00)00079-5 (cit. on p. 11).

[34] Subbarao Kambhampati. “Can Large Language Models Reason and Plan?” In: *Annals of the New York Academy of Sciences* 1534.1 (Apr. 2024), pp. 15–18. ISSN: 0077-8923, 1749-6632. DOI: 10.1111/nyas.15125. arXiv: 2403.04121 [cs]. URL: http://arxiv.org/abs/2403.04121 (visited on 06/08/2025) (cit. on p. 11).

[35] Subbarao Kambhampati et al. *LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks*. June 12, 2024. arXiv: 2402.01817 [cs]. URL: http://arxiv.org/abs/2402.01817 (visited on 11/05/2024). Pre-published (cit. on pp. 11, 77).

[36] Joon Sung Park et al. “Generative Agents: Interactive Simulacra of Human Behavior”. In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*. Ed. by Sean Follmer et al. ACM, 2023, 2:1–2:22. DOI: 10.1145/3586183.3606763. URL: https://doi.org/10.1145/3586183.3606763 (cit. on pp. 12, 13).

[37] Theodore R. Sumers et al. “Cognitive Architectures for Language Agents”. In: *Trans. Mach. Learn. Res.* 2024 (2024). URL: https://openreview.net/forum?id=1i6ZCvflQJ (cit. on p. 12).

[38] Lei Wang et al. “A survey on large language model based autonomous agents”. In: *Frontiers Comput. Sci.* 18.6 (2024), p. 186345. DOI: 10.1007/S11704-024-40231-1. URL: https://doi.org/10.1007/s11704-024-40231-1 (cit. on p. 12).

[39] Sébastien Bubeck et al. “Sparks of Artificial General Intelligence: Early experiments with GPT-4”. In: *CoRR* abs/2303.12712 (2023). DOI: 10.48550/ARXIV.2303.12712. arXiv: 2303.12712. URL: https://doi.org/10.48550/arXiv.2303.12712 (cit. on p. 12).

[40] San Murugesan. "The Rise of Agentic AI: Implications, Concerns, and the Path Forward". In: *IEEE Intell. Syst.* 40.2 (2025), pp. 8–14. DOI: 10.1109/MIS.2025.3544940. URL: https://doi.org/10.1109/MIS.2025.3544940 (cit. on pp. 13, 24).

[41] Hui Yang, Sifu Yue, and Yunzhong He. "Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions". In: *CoRR* abs/2306.02224 (2023). DOI: 10.48550/ARXIV.2306.02224. arXiv: 2306.02224. URL: https://doi.org/10.48550/arXiv.2306.02224 (cit. on p. 13).

[42] Guanzhi Wang et al. "Voyager: An Open-Ended Embodied Agent with Large Language Models". In: *Trans. Mach. Learn. Res.* 2024 (2024). URL: https://openreview.net/forum?id=ehfRiF0R3a (cit. on pp. 13, 77).

[43] Rishi Hazra, Pedro Zuidberg Dos Martires, and Luc De Raedt. "SayCanPay: Heuristic Planning with Large Language Models Using Learnable Domain Knowledge". In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*. Ed. by Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan. AAAI Press, 2024, pp. 20123–20133. DOI: 10.1609/AAAI.V38I18.29991. URL: https://doi.org/10.1609/aaai.v38i18.29991 (cit. on p. 13).

[44] Danny Driess et al. "PaLM-E: An Embodied Multimodal Language Model". In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 8469–8488. URL: https://proceedings.mlr.press/v202/driess23a.html (cit. on p. 13).

[45] Shunyu Yao et al. "ReAct: Synergizing Reasoning and Acting in Language Models". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/forum?id=WE%5C_vluYUL-X (cit. on pp. 13, 22).

[46] Noah Shinn et al. “Reflexion: language agents with verbal reinforcement learning”. In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. Ed. by Alice Oh et al. 2023. URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html (cit. on p. 13).

[47] Alexandre Yukio Ichida, Felipe Meneguzzi, and Rafael C. Cardoso. “BDI Agents in Natural Language Environments”. In: *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024, Auckland, New Zealand, May 6-10, 2024*. Ed. by Mehdi Dastani et al. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024, pp. 880–888. DOI: 10.5555/3635637.3662942. URL: https://dl.acm.org/doi/10.5555/3635637.3662942 (cit. on p. 14).

[48] Minsu Jang et al. “A Structured Prompting based on Belief-Desire-Intention Model for Proactive and Explainable Task Planning”. In: *International Conference on Human-Agent Interaction, HAI 2023, Gothenburg, Sweden, December 4-7, 2023*. ACM, 2023, pp. 375–377. DOI: 10.1145/3623809.3623930. URL: https://doi.org/10.1145/3623809.3623930 (cit. on p. 14).

[49] Alessandro Ricci et al. “The Cognitive Hourglass: Agent Abstractions in the Large Models Era”. In: *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024, Auckland, New Zealand, May 6-10, 2024*. Ed. by Mehdi Dastani et al. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024, pp. 2706–2711. DOI: 10.5555/3635637.3663262. URL: https://dl.acm.org/doi/10.5555/3635637.3663262 (cit. on p. 14).

[50] Sebastian Schmid, Michael Freund, and Andreas Harth. “Adaptive Planning on the Web: Using LLMs and Affordances for Web Agents”. In: *Knowledge Graphs and Semantic Web - 6th International Conference, KGSWC 2024, Paris, France, December 11-13, 2024, Proceedings*. Ed. by Sanju Tiwari et al. Vol. 15459. Lecture Notes in Computer Science. Springer, 2024, pp. 93–108.

DOI: 10.1007/978-3-031-81221-7_7. URL: https://doi.org/10.1007/978-3-031-81221-7%5C_7 (cit. on p. 14).

[51] Takeshi Kojima et al. “Large language models are zero-shot reasoners”. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS 2022)*. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088. URL: https://openreview.net/pdf?id=e2TBb5y0yFf (cit. on p. 22).

[52] Aobo Kong et al. “Better Zero-Shot Reasoning with Role-Play Prompting”. In: *CoRR* abs/2308.07702 (2023). DOI: 10.48550/ARXIV.2308.07702. arXiv: 2308.07702. URL: https://doi.org/10.48550/arXiv.2308.07702 (cit. on p. 22).

[53] Xiangru Tang et al. “Struc-Bench: Are Large Language Models Really Good at Generating Complex Structured Data?” In: *CoRR* abs/2309.08963 (2023). DOI: 10.48550/ARXIV.2309.08963. arXiv: 2309.08963. URL: https://doi.org/10.48550/arXiv.2309.08963 (cit. on p. 23).

[54] Zhi Rui Tam et al. *Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models*. Oct. 2024. DOI: 10.48550/arXiv.2408.02442. arXiv: 2408.02442 [cs]. (Visited on 06/10/2025) (cit. on p. 23).

[55] Brandon T. Willard and Rémi Louf. *Efficient Guided Generation for Large Language Models*. Aug. 2023. DOI: 10.48550/arXiv.2307.09702. arXiv: 2307.09702 [cs]. (Visited on 06/10/2025) (cit. on p. 23).

[56] Jialin Yang et al. *StructEval: Benchmarking LLMs’ Capabilities to Generate Structural Outputs*. May 26, 2025. DOI: 10.48550/arXiv.2505.20139. arXiv: 2505.20139 [cs]. URL: http://arxiv.org/abs/2505.20139 (visited on 06/28/2025). Pre-published (cit. on p. 24).

[57] Do Xuan Long et al. *LLMs Are Biased Towards Output Formats! Systematically Evaluating and Mitigating Output Format Bias of LLMs*. Feb. 23, 2025. DOI: 10.48550/arXiv.2408.08656. arXiv: 2408.08656 [cs]. URL: http://arxiv.org/abs/2408.08656 (visited on 06/28/2025). Pre-published (cit. on p. 24).

[58] Ruslan Kuprieiev et al. *DVC: Data Version Control - Git for Data & Models*. Version 3.60.1. June 2025. DOI: 10.5281/zenodo.15646974. URL: https://doi.org/10.5281/zenodo.15646974 (cit. on p. 55).

[59] Yang Liu et al. *G-Eval: NLG Evaluation Using GPT-4 with Better Human Alignment*. May 23, 2023. DOI: 10.48550/arXiv.2303.16634. arXiv: 2303.16634 [cs]. URL: http://arxiv.org/abs/2303.16634 (visited on 07/04/2025). Pre-published (cit. on p. 55).

[60] Jeffrey Ip and Kritin Vongthongsri. *deepeval*. Version 3.2.3. June 2025. URL: https://github.com/confident-ai/deepeval (cit. on p. 55).

[61] Haitao Li et al. "LLMs-as-Judges: A Comprehensive Survey on LLM-based Evaluation Methods". In: *CoRR* abs/2412.05579 (2024). DOI: 10.48550/ARXIV.2412.05579. arXiv: 2412.05579. URL: https://doi.org/10.48550/arXiv.2412.05579 (cit. on p. 55).

[62] Ari Holtzman et al. *The Curious Case of Neural Text Degeneration*. Feb. 14, 2020. DOI: 10.48550/arXiv.1904.09751. arXiv: 1904.09751 [cs]. URL: http://arxiv.org/abs/1904.09751 (visited on 06/24/2025). Pre-published (cit. on p. 56).

[63] Tobias Kuhn. "A Survey and Classification of Controlled Natural Languages". In: *Computational Linguistics* 40.1 (Mar. 2014), pp. 121–170. ISSN: 0891-2017, 1530-9312. DOI: 10.1162/COLI_a_00168. arXiv: 1507.01701 [cs]. URL: http://arxiv.org/abs/1507.01701 (visited on 06/25/2025) (cit. on p. 77).

[64] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. "Artifacts in the A&A Meta-Model for Multi-Agent Systems". In: *Autonomous Agents and Multi-Agent Systems* 17.3 (Dec. 1, 2008), pp. 432–456. ISSN: 1573-7454. DOI: 10/czjpkd. URL: https://doi.org/10.1007/s10458-008-9053-x (visited on 12/24/2021) (cit. on p. 80).