TU WIEN Informatics

Towards Interoperability between Agentic AI Frameworks through Semantic Representation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Dani Lippmann BSc
Registration Number 12332231

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Fajar J. Ekaputra
Assistance: Dr. Andreas Ekelhart

Vienna, August 14, 2026

Dani Lippmann BSc | Fajar J. Ekaputra

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Declaration of Authorship

Dani Lippmann BSc

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, August 14, 2026

Dani Lippmann BSc

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Fajar Ekaputra, for his continuous support, invaluable guidance, and dedicated oversight throughout the development of this master's thesis.

I am equally grateful to Andreas Ekelhart for his time, insightful feedback, and steadfast support. His oversight and constructive discussions significantly contributed to the successful completion of this work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Agentische KI-Systeme werden zunehmend mit Frameworks wie CrewAI, AutoGen und LangGraph implementiert, die ähnliche architektonische Konzepte jeweils durch grundlegend verschiedene Konstrukte auf Codeebene realisieren. Da jedes Framework diese architektonische Information in seiner eigenen Semantik ausdrückt, lassen sich agentische KI-Systeme über Frameworkgrenzen hinweg nur schwer vergleichen, migrieren oder prüfen. Bestehende semantische Repräsentationen beschreiben agentische Architekturen nur auf hoher Abstraktionsebene und bestehende Transformationsansätze extrahieren sie lediglich in eine Richtung, sodass der vollständige Weg vom Quellcode zu einer semantischen Repräsentation und zurück offen bleibt.

Die vorliegende Arbeit untersucht, inwieweit eine semantische Repräsentation Interoperabilität zwischen modernen agentischen KI-Frameworks ermöglichen kann und verfolgt dies in drei Schritten: Zunächst werden die Anforderungen, die eine solche Interoperabilität an die Repräsentation stellt, aus der Literatur und einer Analyse der drei Zielframeworks erhoben. Anschließend wird eine Ontologie entwickelt, die diese Anforderungen erfüllt. Abschließend wird eine bidirektionale Transformationspipeline erstellt, die Quellcode und Repräsentation verbindet. Zwei Artefakte setzen diesen Ansatz um: Erstens AgentOSCIN, eine OWL-Ontologie, die bestehende Ontologien um sieben Konzeptgruppen erweitert (Agentendefinition, Wahrnehmung und Aufgabendefinition, Koordination, Reasoning, Gedächtnis, Werkzeuge sowie Guardrails und Governance) und die im Anforderungsschritt abgeleiteten Repräsentationsanforderungen kodiert. Zweitens AgenticBIT, eine deterministische Pipeline auf Basis der OSCIN-Methode, die AgentOSCIN-Instanzen mittels statischer AST-Analyse aus Quellcode extrahiert und daraus über Templates frameworkspezifischen Code regeneriert.

Der Ansatz wird anhand eines Korpus von sechs Aufgabenfamilien evaluiert, die über drei agentische KI-Frameworks (d. h. CrewAI, AutoGen und LangGraph) portiert wurden. AgentOSCIN beantwortet alle 35 Kompetenzfragen vollständig, gegenüber 22,9 Prozent bei AgentO und 17,1 Prozent bei einem vereinheitlichten Klassenmodell als Baseline. Innerhalb eines Frameworks erreicht die Extraktion einen mittleren Triple F1 von 0,986 und die Generierung einen mittleren AST F1 von 0,882 und übertrifft damit eine LLM-basierte Extraktionsbaseline deutlich. Über Frameworkgrenzen hinweg bleibt strukturelle Information nur teilweise erhalten (mittlerer Aligned F1 von 0,569) und die generierten Systeme sind ohne Unterstützung selten lauffähig; ein LLM-basierter Reparaturschritt

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

steigert die vollständige Zielerreichung von 3 von 54 auf 29 von 54 generierte Systeme. Die Ergebnisse zeigen, dass eine semantische Repräsentation die Architektur agentischer Systeme über Frameworkgrenzen hinweg tragen kann, während ausführbare Interoperabilität dadurch begrenzt bleibt, was die Repräsentation zu modellieren wählt.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Agentic AI systems are increasingly built with frameworks such as CrewAI, AutoGen and LangGraph, each of which realises similar architectural concepts through fundamentally different code-level constructs. Because each framework expresses this architectural information in its own semantics, agentic AI systems are hard to compare, migrate or audit across frameworks. Existing semantic representations describe agentic architectures only at a high level, and existing transformation approaches extract them in one direction only, leaving the round trip from source code to a semantic representation and back as open work.

This thesis investigates to what extent a semantic representation can enable interoperability between modern agentic AI frameworks and pursues this in three steps: it first elicits the representational requirements such interoperability demands, from the literature and an analysis of the three target frameworks; it then develops an ontology that meets them; and it finally builds a bidirectional transformation pipeline that connects source code and representation. Two artefacts realise this approach: First, AgentOSCIN, an OWL ontology, extends existing ontologies across seven concept groups (agent definition, perception and task, coordination, reasoning, memory, tools, and guardrails and governance) and encodes a set of representational requirements derived in the requirements step. Second, AgenticBIT, a deterministic pipeline based on the OSCIN method, extracts AgentOSCIN instances from source code via static AST analysis and regenerates framework-specific code from them via templates.

The approach is evaluated over a corpus of six task families ported across three Agentic AI frameworks (i.e. CrewAI, AutoGen and LangGraph). AgentOSCIN answers all 35 competency questions fully, against 22.9 per cent for AgentO and 17.1 per cent for a unified class model baseline. Within a framework, extraction reaches a mean Triple F1 of 0.986 and generation a mean AST F1 of 0.882, clearly outperforming an LLM-based extraction baseline. Across framework boundaries, structural information is preserved only partially (mean Aligned F1 of 0.569) and generated systems rarely run unaided; an LLM-based repair step raises full goal attainment from 3/54 to 29/54 generated systems. The results show that a semantic representation can carry the architecture of agentic systems across frameworks, while executable interoperability remains bounded by what the representation chooses to model.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**

2 Related Work **5**
2.1 From Multi-agent Systems to Agentic AI 5
2.2 Semantic Representations for Agentic AI Systems 7
2.3 Ontology Engineering Approaches 8
2.4 Model-based Transformation Methods 9
2.5 Research Gap 9

3 Research Methods **11**
3.1 Requirement Elicitation through Literature Review and Framework Analysis 12
3.2 Ontology Engineering using Ontology Development 101 13
3.3 Bidirectional Transformation using the OSCIN Method 13
3.4 Evaluation 15

4 Requirement Elicitation **17**
4.1 Literature Study: Deriving the Core Concepts of Agentic AI Systems . 17
4.2 An Analysis of Agentic AI Frameworks 23
4.3 Results of Requirements Elicitation 34

5 AgentOSCIN Ontology **37**
5.1 Scope and Competency Questions 37
5.2 Ontology Reuse 38
5.3 Defining Classes and Properties 42

6 AgenticBIT: Agentic AI Bidirectional Transformation Framework **53**
6.1 Design Principles 53

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.2 Extraction: From Source Code to Ontology 54
6.3 Generation: From Ontology to Source Code 57

7 Evaluation **61**
7.1 Ontology Evaluation 61
7.2 Transformation Experiments 71

8 Result Analysis **89**
8.1 Answers to Research Questions 89
8.2 Limitations 93

9 Conclusion and Outlook **95**

Overview of Generative AI Tools Used **97**

List of Figures **99**

List of Tables **101**

Bibliography **103**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

Over the past several years, agentic AI systems have moved from research prototypes to real world production deployments, driven by organisational demand to automate complex tasks without continuous human oversight [BKN+25]. In contrast to conventional AI systems, agentic systems are capable of planning, delegating and acting autonomously across extended tasks [AKA25]. However, as these systems scale to handle more sophisticated and distributed workflows, their architectures become increasingly complex, exposing severe challenges in interoperability, scalability as well as system integration [DBM25].

This is particularly critical as agentic AI systems, in practice, are frequently developed in a monolithic and ad hoc manner, relying on framework-specific abstractions and tightly coupled implementations [SRK26, EKEK26]. Frameworks such as LangGraph [1], CrewAI[2] and AutoGen[3] each realise similar agentic concepts through fundamentally different code-level constructs (e.g. graph-based state machines, role-based agent teams and conversation-driven orchestration). Architectural decisions and coordination logic are often hard-coded into source code, thereby limiting modularity and transferability across systems [EKEK26].

This lack of standardized and transferable architectural semantics has been identified as a central challenge for agentic AI, as it hinders the comparison of implementations, the replication of results and generalization across frameworks [SRK26]. Additionally, researchers emphasize the critical need to decouple an agent's semantic reasoning from its structural coordination logic [BBP25]. In current framework implementations, these two dimensions are frequently conflated, resulting in systems that rely on opaque, prompt-dependent routing mechanisms to handle both cognitive tasks and process orchestration. This architectural entanglement not only hinders the formal verification of execution

[1] https://langchain-ai.github.io/langgraph/
[2] https://www.crewai.com/
[3] https://microsoft.github.io/autogen/

pathways, but it also introduces severe scalability bottlenecks by forcing the system to invoke expensive language model inferences for routine, deterministic state transitions [BBP25].

A structurally deep representation is ultimately a basic requirement for a robust "Web of Agents" [PBPS25]. It also enables the seamless conversion of agentic workflows from one framework to another [EKEK26], supports the formal auditing of system safety from a systems-theoretic view, and reduces emergent systemic risks without losing the adaptive reasoning power of the underlying models [MRV+25].

Existing semantic representations partially address these challenges by providing expressive conceptual vocabularies for agents, tasks and workflows [DBM25, EKEK26]. More recent work such as AgentO additionally demonstrates the feasibility of extracting agentic workflows from framework-specific source code into a unified ontology [EKEK26] and thereby showing operational usability of such representations.

However, these approaches remain focused on unidirectional extraction and do not model constructs such as conditional control flow, state management, or prompt-level semantics at sufficient depth to support bidirectional transformation between frameworks. This unidirectional limitation shows that existing representations are not yet expressive enough to serve as a shared intermediate layer between modern agentic AI frameworks and therefore cannot support the systematic migration or comparison of agentic architectures across different code implementations.

This thesis aims to contribute to addressing the challenges above and enrich the discourse on the topic of agentic AI systems and their semantic representation. Specifically, it aims to answer the main research question:

> *To what extent can a semantic representation enable interoperability between modern agentic AI frameworks?*

Answering this question requires demonstrating that the vocabulary is rich enough to carry the architectural information of agentic AI systems in both directions. This means from framework-specific implementations into a formal representation and from that representation back into executable code. The thesis approaches the question through round-trip transformation, using the quality of extraction and regeneration as a measurable indicator of how far a shared ontology can close the gap between frameworks. To structure this research, the main question is split into four sub-questions:

- **RQ1:** What are the requirements for an ontology to facilitate interoperability between modern agentic AI frameworks?
- **RQ2:** To what extent does the proposed ontology satisfy the requirements for agentic AI interoperability?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **RQ3:** To what extent can an ontological representation be systematically extracted from an agentic AI system's source code?
- **RQ4:** To what extent can an agentic AI system's source code be systematically generated from its ontological representation?

This thesis addresses these research questions through a three-step approach. First, it identifies the requirements a semantic representation must satisfy to make agentic systems interoperable, drawing on both the literature and an analysis of CrewAI, AutoGen and LangGraph. Second, it develops the AgentOSCIN ontology, a framework-independent representation that meets these requirements. Third, it designs and develops AgenticBIT, a deterministic bidirectional source-code-to-ontology transformation pipeline that extracts this representation from source code and regenerates executable code from it.

The two artefacts (i.e. AgentOSCIN and AgenticBIT) are evaluated together across competency question coverage, extraction quality metrics, generation quality and cross-framework transfer to assess how far the semantic representation supports interoperability between modern agentic AI frameworks. Additionally, the runtime of the generated source code is evaluated, to assess whether the generated code produces the expected outputs.

The rest of the thesis is divided into the following chapters. First, we discuss and elaborate the related work in Chapter 2. Following the related work, the research structure and the approaches taken to answer the research questions are set out as the research methodology in Chapter 3. After that the requirements for an interoperability ontology are elicited and synthesised in Chapter 4. Then the design of AgentOSCIN ontology and the engineering decisions behind it are described in Chapter 5. Following that, the bidirectional source code to ontology transformation framework, its architecture and features are detailed in Chapter 6. Chapter 7 then reports the evaluation of both artefacts. Subsequently, the implications of this work are discussed and the limitations are shown in Chapter 8. Finally, the thesis concludes with a research summary and a future outlook (see Chap. 9).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Related Work

This chapter surveys the work this thesis builds on and positions the contributions of this thesis within the existing landscape. The chapter is organised into five sections. Section 2.1 covers multi-agent systems and the three agentic AI frameworks analyzed in this thesis. Section 2.2 reviews semantic representations proposed for AI systems, with particular attention to ontologies that capture agentic concepts. Section 2.3 introduces the ontology engineering methodologies that guide the development of AgentOSCIN. Section 2.4 covers model-based transformation approaches. Section 2.5 identifies the gap this thesis addresses.

2.1 From Multi-agent Systems to Agentic AI

Multi-agent systems are not a new research field with work on coordinated autonomous agents dating back decades [WJ95, Woo09]. Classical MAS research was driven by questions of distributed problem solving, concurrent rationality and open systems. The emergence of Large Language Models (LLMs) did not change these underlying problems; it shifted where research attention sits. LLMs have become the core building block of agents, which are now natural-language reasoners with broad general competence rather than narrow symbolic planners. The coordination problem is still present, but its treatment has shifted from symbolic message exchange to prompt engineering, tool orchestration and workflow control [GCW+24, VRB26].

Agentic AI did not emerge from a single predecessor field but from the convergence of several. As shown in Figure 2.1, it draws on LLMs for natural-language reasoning and planning, on reinforcement learning for adaptive goal-directed behaviour and on MAS for the coordination principles that govern how multiple agents interact within a shared environment [BKN+25]. Coordinating multiple agents to achieve complex objectives is both the defining feature of MAS and the operational core of modern agentic AI frameworks. Where classical MAS relied on symbolic message exchange, agentic AI reuses

those coordination structures while replacing the reasoning substrate with foundation models [Bot25].

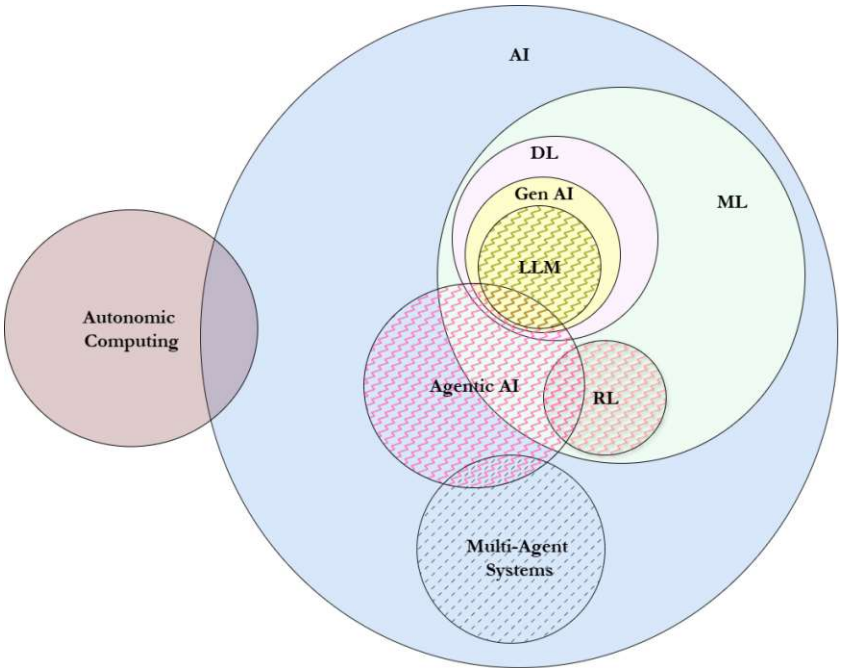

Figure 2.1: Conceptual foundations of agentic AI [BKN+25].

The transition towards LLM-based multi-agent systems was motivated in large part by tasks that exceed the context window, domain coverage or reasoning budget of any single model [VRB26]. Early LLM-based frameworks explored different communication topologies to manage this coordination, categorizing interactions into chain (waterfall), star (hub-and-spoke) and mesh (swarm) patterns [VRB26]. Among these, three modern agentic AI frameworks have become influential in both research and production settings: AutoGen, CrewAI and LangGraph [Bot25, DBM25, EKEK26]. Each realises multi-agent collaboration through a different architectural commitment.

AutoGen treats multi-agent collaboration as a conversation between conversable agents. Its underlying model is dialogical. This means that agents, tools and humans are all first-class participants in a shared message stream and the system's behaviour emerges from the rules that govern who speaks next [WBZ+23]. CrewAI treats multi-agent collaboration as role-based teamwork. Agents are defined by personas and responsibilities, organised into crews and coordinated through processes that resemble organisational structures [Cre25]. LangGraph treats multi-agent collaboration as a programmable workflow. Its underlying model is computational. The system is a state machine whose transitions are explicit and whose execution is traceable. Agent identity is secondary to graph structure [Lan25, Cla26].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2.2 Semantic Representations for Agentic AI Systems

Alongside the development of agentic AI frameworks, several semantic representations have been proposed to capture the structure and interactions of complex AI systems. A first line of work develops ontological representations of AI systems from the boxology design-pattern framework, which is a graphical methodology used to conceptualize, design and analyze hybrid AI systems [HT19]. EASY-AI extends this boxology by pairing its visual notation with an underlying ontology that supplies the standardization and axiomatic constraints governing how the components of an AI system may be combined [EDS+24]. Building on this foundation, the BEAM ontology provides a high-level semantic model for representing AI systems within their operational and organisational context, grounding its core classes in EASY-AI and adding concepts for systems, resources, containers and engineering annotations together with a dedicated perspective on AI system risks and mitigation [EPK25].

These approaches demonstrate the value of semantic representations for documenting and comparing AI systems, but they all operate at the level of neuro-symbolic system architecture and remain largely agnostic to the fine-grained agentic control structures and execution semantics required to model multi-agent agentic AI systems.

A broader comparative perspective on the agentic AI landscape is offered by Derouiche et al. [DBM25]. They survey leading agentic AI frameworks and analyze their architectural principles, communication mechanisms, memory management and guardrail support. Rather than treating each framework in isolation, they combine their recurring structural components into a single unified class model (see Fig. 2.2), in which an agent reasons through a prompt, uses tools and memory, is assigned tasks and is constrained by guardrails, with the LLM acting as the shared reasoning engine across these elements.

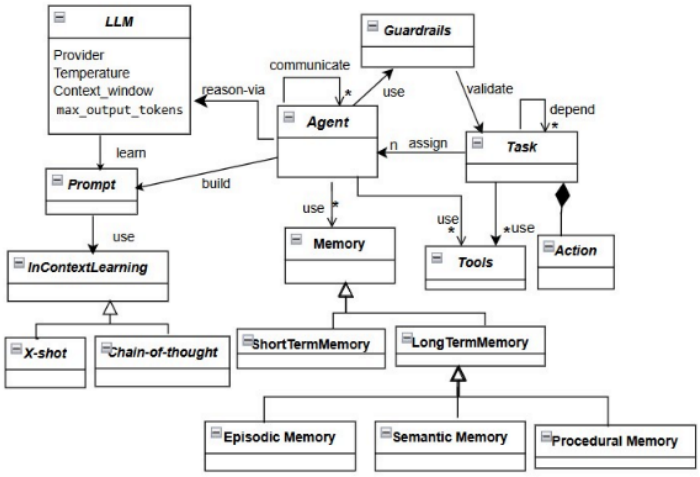

Figure 2.2: Unified class model showing the core components of agentic AI systems [DBM25]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Another recently developed semantic representation that serves as foundational work for this thesis is AgentO. It is an ontology designed specifically for agentic AI systems [EKEK26]. The central premise is that contemporary agentic frameworks lack a shared conceptual model as each framework encodes its logic directly in code. This results in monolithic implementations that hinder reusability, interoperability and auditing across systems. AgentO addresses this by offering a standardized RDF/OWL vocabulary for the recurring concepts of the architecture (agents, goals, tasks, tools, resources, workflows and their interrelations) so that agentic AI systems can be represented declaratively and independently of any particular framework [EKEK26].

2.3 Ontology Engineering Approaches

Ontologies are able to capture implicit knowledge from heterogeneous data sources and achieve semantic interoperability [ESS+17, WVV+02]. By mapping data sources into a domain ontology, data consumers can query a single unified vocabulary without any prior knowledge of how the underlying data is organised [KHS+17]. The mappings specify how the ontology is populated, the ontology provides the vocabulary and the consumers interact only with that shared layer [KHS+17]. This core concept of ontology-based data access guides the central goal of this thesis: providing a shared semantic representation that unifies the design of agentic AI systems across frameworks and thereby enables a move toward framework-level interoperability.

The engineering of such representations is supported by a range of established methodologies [EEQ22]: From comprehensive frameworks such as METHONTOLOGY [FLGPJ97], the NeOn methodology [GPSF09], and the Basic Formal Ontology (BFO) methodology [ASS15], to more agile approaches such as AMOD [ARH19]. This thesis draws on two further agile, iterative methodologies, Ontology Development 101 and the Linked Open Terms (LOT) methodology, which are the most relevant here and are described in more detail below.

Ontology Development 101 provides a lightweight, iterative guide for building domain ontologies [NM01]. It organises the process around enumerating the important terms of a domain, defining classes and their hierarchy, specifying properties and creating instances and recommends using competency questions to bound the scope of the ontology. It puts emphasis on iteration over completeness, which makes it well suited to domains where the vocabulary is still stabilizing, as is the case for agentic AI systems. The LOT methodology, developed by Poveda-Villalón et al. [PVFIFLGC22], targets industrial and agile software engineering contexts. It structures development around a full lifecycle of requirements specification, implementation, publication and maintenance and places explicit emphasis on the reuse of existing vocabularies and on producing ontologies that are ready for open publication. Where Ontology Development 101 prioritises getting a working ontology quickly, LOT adds the governance and publication discipline expected in production settings.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

This thesis follows Ontology Development 101 for the development of AgentOSCIN, as its iterative structure fits better to the evolving requirements of agentic AI systems.

2.4 Model-based Transformation Methods

In the context of agentic AI systems, recent work such as AgentO demonstrates the feasibility of transforming agentic AI source code from framework-specific implementations into a unified semantic representation using LLMs [EKEK26]. By providing the LLM with their ontology and framework-specific source code, the model successfully produced a valid instance of the ontology containing the source code patterns. This approach enables declarative reconstruction, cross-framework comparison and workflow auditing. However, LLM-based approaches are non-deterministic, as even when they are configured with deterministic settings, they can still produce different outputs across repeated runs [AAC+25]. Their results can also vary across model versions, which makes the same extraction process less stable over time [PCR+24]. These limitations make it hard to compare and evaluate the quality and correctness of such LLM-based approaches.

Model Driven Engineering (MDE) establishes the theoretical basis for addressing this limitation. In MDE, models are treated as primary development artefacts from which executable code is deterministically generated rather than as just documentation [SRF18, FC18]. This has been applied to many domains (e.g. Software Engineering [FC18]), demonstrating that a shared model can serve as an intermediary between heterogeneous implementations in both directions.

A modern and ontology-based instantiation relevant to the present work is the Ontology-based Source Code Interoperability (OSCIN) method [dAZS21]. OSCIN separates syntactic from semantic abstraction. Source code is first parsed into a syntactic structure specific to its host language and then mapped to a shared ontology representing a common subdomain. By routing heterogeneous source languages through a central operational ontology, OSCIN mitigates both syntactic and semantic heterogeneity and exposes a single abstraction layer. OSCIN applies this layer to analytical operations such as querying and auditing. However, the method does not provide mechanisms to regenerate executable, framework-specific implementations from an ontology.

The unidirectionality gap therefore still persists at the method level. This thesis addresses it by extending OSCIN's application perspective with a code generation direction and treating the populated ontology not as an analytical artefact but as a source from which framework-specific source code can be reconstructed.

2.5 Research Gap

Existing semantic representations capture AI systems at a high architectural level but do not model the finer control logic, such as coordination and reasoning flow, that distinguishes agentic AI systems. AgentO supplies a framework-independent vocabulary

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

but demonstrates only a single, non-deterministic extraction into it [EKEK26]. What remains unknown is whether such a vocabulary can be made rich enough to capture the orchestration, coordination, reasoning, memory and guardrail concepts that vary across frameworks [DBM25].

A second gap concerns operational usability. An expressive representation does not by itself establish that it can be used in practice [SWMG+22]. Even when a vocabulary is rich enough on paper it can still lack a reliable, reproducible way to recover real systems from code and regenerate them. Current transformation approaches map source code to a semantic representation in one direction only and can describe or extract a workflow but not regenerate it [dAZS21, EKEK26]. The round trip from code to representation and back is left as open future work [EPK25, EKEK26].

This thesis addresses these gaps by first extending existing semantic representations into such a representation and then pairing it with a deterministic round trip that extracts an implementation into the ontology and regenerates one from it.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Research Methods

This research follows the Design Science Research (DSR) approach as defined by Hevner et al. [HMPR04] and Hevner [Hev07]. Requirement elicitation addresses the relevance cycle, ontology engineering bridges the rigor and design cycles and the bidirectional transformation method together with its evaluation completes the design cycle.

The concrete instantiation of DSR for this thesis is shown in Figure 3.1. Requirements for agentic AI interoperability are first elicited from a study of the relevant literature and the modern agentic AI frameworks CrewAI, LangGraph and AutoGen. These requirements then drive the design of the AgentOSCIN ontology, following Ontology Development 101 [NM01] and reusing existing ontologies (e.g. AgentO [EKEK26], DBM [DBM25], BEAM [EPK25]). The ontology then grounds the OSCIN-based transformation framework (i.e. AgenticBIT), with the ontology and the framework each evaluated in a dedicated step.

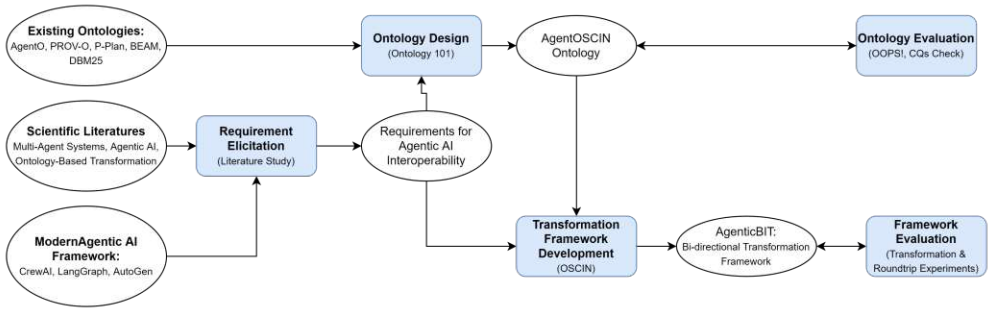

Figure 3.1: Methodology Overview

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.1 Requirement Elicitation through Literature Review and Framework Analysis

The requirement elicitation phase aims to identify representational requirements for a semantic model of agentic AI systems that supports bidirectional transformation. The phase is structured in two steps.

The first step is a literature review conducted according to Grant and Booth [GB09]. The review follows a structured process of identifying relevant sources, extracting recurring architectural concepts and organizing them into a cross-paper concept matrix (see Chap. 4). The search string ("agentic AI" OR "multi-agent systems") AND ("ontology" OR "representation" OR "architecture") was applied to Google Scholar and Semantic Scholar with a cut-off date of April 15, 2026.

The research field is new and rapidly evolving. For this reason preprints were included alongside peer-reviewed publications. A paper was included if it (i) addressed the architectural design of LLM-based or classical multi-agent systems, (ii) was written in English and (iii) was accessible in full text. A paper was excluded if it (i) focused on single-agent systems without multi-agent considerations, (ii) treated multi-agent systems purely as an application case study rather than analyzing their design, (iii) concentrated only on agent communication protocols or runtime behaviour rather than on architectural structure. Classical multi-agent systems references such as Wooldridge's foundational work (i.e. [WJ95, Woo09]) were included as conceptual anchors for the taxonomy.

The retained papers were then analyzed to extract the architectural concepts each paper identifies as important for agentic AI systems. Concepts were coded individually per paper and then aggregated across the whole corpus. Semantically similar concepts were grouped into higher-level categories, which form the main concepts of the taxonomy.

The second step is an analysis of framework-specific documentation. The concepts identified in the literature review are examined at the source code level across the documentation of three agentic AI frameworks that represent fundamentally different architectural paradigms: sequential role-based (CrewAI), graph-based (LangGraph) and conversation-driven (AutoGen). For each concept, the analysis documents how and whether the concept is realised in each framework. Developer documentation [Cre25, Mic25, Lan25] and selected open-source code bases of the three frameworks are the primary sources. The specific versions analyzed are CrewAI 1.13.0, AutoGen (`autogen-agentchat`) 0.7.5 and LangGraph at the state of its main branch as of April 20, 2026.

The output of this step is a set of requirements that a framework-agnostic ontology must fulfill in order to represent modern agentic AI systems. Each requirement is traceable back to at least one framework-level construct, one literature concept or both.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.2 Ontology Engineering using Ontology Development 101

The second methodological step is the development of a formal semantic representation for agentic AI systems. This thesis proposes the development of a framework-agnostic ontology to semantically represent the requirements of phase one. Ontology engineering is conducted following the Ontology Development 101 methodology [NM01] and is applied in four iterative activities.

First, the scope and domain of the ontology are determined based on the concepts and requirements list produced in the previous phase. The scope is bounded by the three target frameworks and the agentic AI concepts identified across them. Second, competency questions are formulated to define what the ontology must be able to answer. These questions are derived directly from the requirements elicitation and serve both as design guidance during modelling and as an evaluation criterion during evaluation. Third, existing vocabularies are evaluated for reuse. AgentO [EKEK26] is adopted as the primary starting point because it already covers the recurring concepts of the agentic paradigm and builds on established upper vocabularies (PROV-O, P-Plan and BEAM). Additional data models (e.g. unified class model by Derouiche et al [DBM25]) are also analyzed to gather additional vocabulary for our ontology. Concepts or properties that are not expressible by existing literature, as identified through the requirement analysis, are introduced as AgentO extensions. This includes framework-specific constructs required for bidirectional transformation. Fourth, the extended ontology is formalized in OWL to ensure machine interpretability and compatibility with existing Semantic Web standards. The resulting ontology is referred to as AgentOSCIN (Agentic AI Ontology for Source Code Interoperability).

3.3 Bidirectional Transformation using the OSCIN Method

The third methodological step is the construction of a bidirectional transformation framework that links framework-specific source code patterns of agentic AI systems to AgentOSCIN. The method follows the five phases of the OSCIN method for ontology-based source code interoperability (see Fig. 3.2).

The first phase (i.e. specification) defines the scope of the transformation. This is achieved by step one of this thesis (see Chap. 4), where the output is the requirement specification table that guides the ontology engineering. The second phase, Subdomain Semantic Abstraction, is the construction of the domain ontology and is addressed by the ontology engineering step above. The third phase, Language Syntactic Abstraction, defines how source code is parsed into syntactic structures that the transformation can operate on. The fourth phase, Language Semantic Abstraction, maps these syntactic structures to the corresponding instances of our semantic representation (i.e. AgentOSCIN) [dAZS21].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

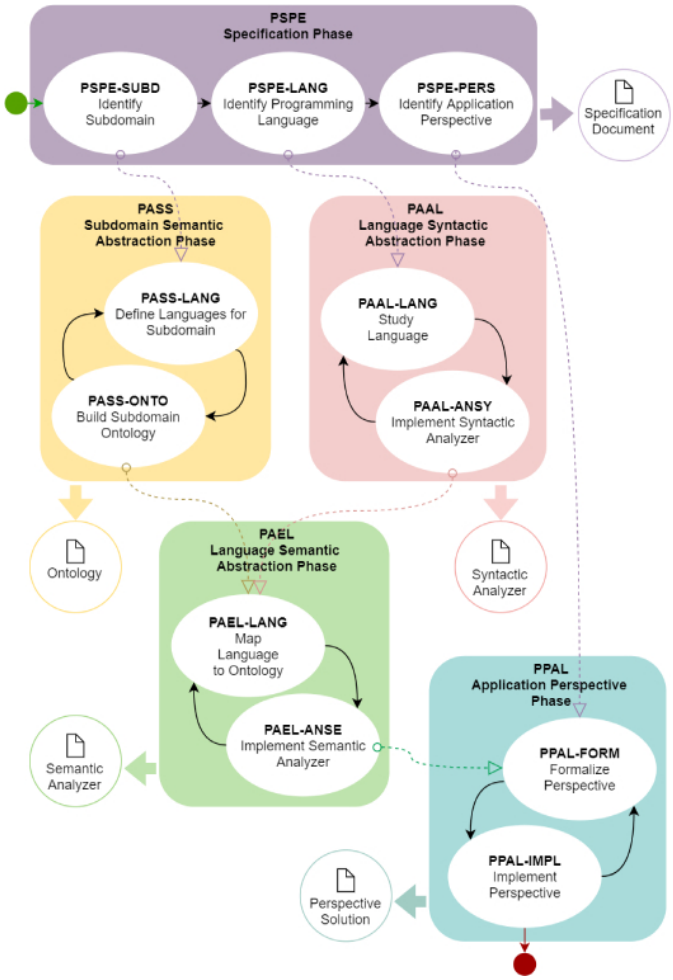

Figure 3.2: OSCIN Method Overview [dAZS21]

This is done using Python AST modules [Pyt24] to extract frameworks-specific source code details which can then be mapped to the semantic representation.

The fifth phase, Application Perspective, specifies what is done with the populated ontology. The original OSCIN method applies this phase to analytical operations over the semantic graph such as querying, comparison and auditing [dAZS21]. This thesis implements the fifth phase with code generation. The populated ontology is treated not as an analytical artefact but as a source from which framework-specific source code can be regenerated. This extension is what makes the transformation bidirectional.

The transformation pipeline is deterministic. It operates through static parsing and mapping rules. Constructs that cannot be resolved without interpretation of natural-language content or semantic analysis of source code are identified but excluded from the pipeline, in order to provide reproducibility. The resulting gaps in coverage are reported as part of the evaluation (see Chap. 7) and discussed as limitations and directions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

for future work (see Chap. 8 and Chap. 9). The technical realisation of the pipeline, including parser implementation, intermediate data structures and code generation, is presented in Chapter 6.

3.4 Evaluation

The fourth and final methodological step evaluates the two artefacts produced by the preceding steps. The evaluation is structured along two dimensions that reflect their interdependent nature. The first assesses AgentOSCIN as a semantic representation in its own. The second assesses the OSCIN-based transformation method that operationalizes the ontology. This tests whether the ontology carries enough structure to support extraction and generation. The detailed process and results are reported in Chapter 7.

3.4.1 Ontology Evaluation

The first dimension evaluates the structural quality of AgentOSCIN and its coverage of the elicited requirements. It consists of two steps.

- **Structural Evaluation with OOPS!:** The ontology is first evaluated with the OOPS! Pitfall Scanner [PVGPSF14]. OOPS! inspects an ontology against a catalogue of common modelling pitfalls such as missing inverse relationships, unconnected elements and inconsistent naming conventions and flags each detected issue by severity as critical, important or minor.
- **Competency Question Coverage:** The second step examines whether the ontology captures enough of the architectural concepts required for cross-framework representation. This follows the competency coverage evaluation described in the Ontology Development 101 methodology [NM01]. For each competency question defined during ontology engineering, it is checked whether the question can be answered from the structure the ontology provides, expressed as a SPARQL query over a populated ABox. To quantify the expressiveness gained over existing work, the same competency questions are evaluated against AgentO [EKEK26] and the unified class model by Derouiche et al.[DBM25]. AgentO serves as the primary baseline because it is the only existing ontology designed specifically for agentic AI systems and is the direct starting point for AgentOSCIN. Each question is scored on a three-level metric. Coverage is *full* when all concepts required to answer the question are representable in the model, *partial* when some but not all required concepts are representable and *none* when the model provides no means of representing the concepts the question asks about.

3.4.2 Transformation Experiments

The second dimension uses controlled transformation experiments to assess the operational usability of AgentOSCIN and AgenticBIT. Five experiments are run over the corpus.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Each is summarised here and described in full in the evaluation chapter.

- **Extraction:** The extraction experiment runs the extractor on the source code corpus and compares the resulting ABox against the hand-authored ground-truth for that same cell, measuring how faithfully the deterministic pipeline recovers the intended semantic representation.
- **LLM Extraction Baseline:** To put the extraction numbers in context an LLM baseline is run alongside the deterministic extractor, in which an LLM is given the same source code and the full ontology in context and asked to produce the ABox directly. The baseline is evaluated with the same extraction metrics and provides a comparison point for the deterministic approach.
- **Generation:** The generation experiment runs the generator on the hand-authored ground-truth ABox of a cell and emits source code for the same target framework, which is then structurally compared against the original source code with the source code metric suite.
- **Cross-Framework Round-Trip:** The cross-framework round-trip experiment feeds the hand-authored ABox of a family in source framework S to the generator with target framework $T \neq S$. The generated target-framework source tree is re-extracted with the extractor and the resulting ABox is compared against the hand-authored target-framework ground-truth for the same family. The generated code is additionally compared against the hand-authored target source. This experiment tests whether the semantic representation transfers across frameworks.
- **Runtime Executability:** The runtime executability experiment runs all generated code from the generation and cross-framework experiments in isolation and compares the runtime output against the expected output of the agentic AI system. This tests whether enough information is preserved through the transformation to still produce the expected results, or only structurally plausible code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Requirement Elicitation

This chapter aims to answer the first research question of this thesis, namely what the requirements for agentic AI interoperability are. The question is answered through a literature review in two separate steps. The first step is a literature study to derive the core concepts a semantic representation needs to model in order to represent agentic AI systems. These core concepts then act as the starting point for the framework analysis, which examines how modern agentic AI frameworks concretely implement them. Here we consulted the official framework documentation of CrewAI, AutoGen and LangGraph and analysed it with respect to each concept. For each concept we then derive a set of requirements, traceable back to the scientific literature and the framework documentation, to provide a reproducible starting point for the ontology engineering phase.

4.1 Literature Study: Deriving the Core Concepts of Agentic AI Systems

The aim of this phase is to produce a conceptual baseline for agentic AI systems. Without it, the following framework analysis would lack criteria for deciding which aspects of each framework need closer attention at the source code level. This is important given that there is currently no single agreed-upon data model that categorizes these designs in the research community. Foundational work on MAS established core notions such as reasoning, agent interaction, coordination and communication [Woo09] and recent studies have updated these classical ideas for LLM-based agents, adding elements such as tool use, memory architectures and reasoning patterns [EKEK26, DBM25, BKN+25].

However, these contributions differ in scope and vocabulary. Some propose unified class models of an entire agentic AI system [DBM25], while others reduce the system to a small number of pillars such as reasoning, memory and tool use [Bot25, SRK26]. No common reference model exists against which the three target frameworks could be analyzed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

In the absence of such a model, the core concepts were derived from the corpus and not adopted from any single source. The process was structured as follows. Each source in Table 4.1 names, in some form, the architectural concepts of agentic AI or multi-agent systems. For every source, the architectural concepts it names were extracted and recorded. These concepts were then mapped across the corpus to identify which concepts recurred. Through this mapping, seven concept areas emerged as the ones that consistently reappeared across the literature. These seven recurring concepts are Agent Definition, Perception and Task, Coordination, Reasoning, Memory, Tools, and Guardrails and Governance.

4.1.1 Agent Definition

Wooldridge and Jennings provide one of the earliest comprehensive surveys of agent theories, architectures and languages, establishing core properties such as autonomy, reactivity, proactivity and social ability as defining characteristics of an agent. They define agents as a computing system situated in an environment that is capable of flexible, autonomous action to meet its design objectives [WJ95].

In modern LLM-based multi-agent systems, the agent remains a central architectural component, but its characterization has changed from simple reactive/proactive agents to fully self-aware agents, that exhibit meta-cognitive awareness [NSSP26]. Additionally, an agent is based on a foundation model (like an LLM) and pursues complex, multi-step goals with limited direct supervision [Sch25]. These agents are often defined with specific profiles, roles, backgrounds and constraints that specify how they act within the system [LWZ+24] [GCW+24].

4.1.2 Perception and Task Definition

The perception layer is responsible for ingesting external inputs from the environment and translating them into internal representations that the agent can reason about [BKN+25]. Earlier foundation models relied almost exclusively on text, but modern agentic systems are increasingly equipped with multi-modal perception capabilities. They can now process inputs including textual descriptions, visual images, auditory signals and spatial coordinates [LWZ+24] [WZ25].

Critically, agentic systems do not rely on rigid, explicit commands. They act upon abstract, high-level user goals. The perception layer translates these goals, alongside multi-modal context, into structured internal representations that drive the agent's behaviour [BKN+25]. It therefore defines the boundary at which human intent enters the agent system and takes on a computationally tractable form. In the broader agentic AI literature, the perception layer describes how agents ingest and internalize external input at runtime. At the design-time abstraction level addressed by this thesis, the concern is how developers specify the tasks and actions an agent system should perform [DBM25].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.1.3 Coordination

When tasks exceed the capacity of a single agent, Agentic AI employs MAS, where specialized agents coordinate their efforts to solve complex problems [Bot25]. The architecture of this coordination typically falls into several structural categories. Centralized orchestration, where a single supervisor agent decomposes tasks and routes them to sub-agents. Hierarchical structures, where agents operate in a tree-like command flow and decentralized or peer-to-peer networks, where agents negotiate and collaborate directly without a central authority [BKN+25] [LWZ+24].

Beyond structural organisation, the mechanisms of coordination are defined by the interaction patterns among agents. These can be either cooperative, adversarial (debate), or mixed [LWZ+24]. In cooperative teamwork, agents share information and divide labor to reach a common goal. Adversarial debate involves agents presenting and defending conflicting viewpoints to cross-validate reasoning, reduce hallucinations and refine solutions [GCW+24].

To efficiently manage the flow of information during these collaborative processes, systems often utilize a Shared Message Pool, a publish-subscribe mechanism allowing agents to extract only the information relevant to their specific roles without requiring direct point-to-point communication [HZC+24].

4.1.4 Reasoning

The reasoning engine acts as the cognitive core of the agent, transforming high-level goals into actionable steps through multi-step reasoning and planning [LWZ+24]. In contrast to traditional models that provide immediate, single-step responses, agentic AI performs a step-by-step solution process that includes problem analysis, planning, solving, reflection and validation [Sch25].

To implement this robust cognition, systems employ structured reasoning frameworks such as Chain of Thought , Tree of Thoughts and Graph of Thoughts [LWZ+24]. These architectural frameworks allow agents to explore multiple partial reasoning chains concurrently, evaluate the viability of intermediate results and perform self-correction before committing to a final action [Sch25]. Furthermore, reasoning processes can be categorized into one-step decision-making, where a task is decomposed and executed in a single logical flow and multi-step reasoning, which requires iterative reasoning cycles that maintain long-term consistency with the overall objective [LWZ+24, GCW+24].

4.1.5 Memory

Memory provides agents with temporal continuity and the foundational knowledge required to navigate complex environments [BKN+25]. The architecture is typically divided into distinct functional durations. Short-term memory (or working memory) maintains the immediate conversational or task context within the language model's context window constraints [DBM25] [Sch25]. Long-term memory, on the other hand,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

captures persistent data across sessions, allowing the agent to recall historical interactions, user preferences and learned knowledge over time [DBM25].

Beyond duration, long-term memory can be functionally categorized into semantic memory for storing facts and reasoning paths, procedural memory for recalling specific task flows and strategies and episodic memory for retaining detailed contextual snapshots of specific past events [DBM25]. To manage and access this amount of information, agents frequently utilize Retrieval-Augmented Generation (RAG) and external vector databases. These systems retrieve relevant text snippets based on semantic similarity and append them to the agent's prompts, thereby reducing hallucinations and supporting dynamic learning [GCW+24] [Sch25]. Memory operations also involve active reflection, where agents summarise and distill past experiences to continually enhance their cognitive abilities and adaptability [LWZ+24].

4.1.6 Tool Integration

Tool integration can significantly expand an agent's capabilities beyond its static, pre-trained knowledge by enabling it to interact directly with external systems and physical or digital environments [Sch25]. By dynamically invoking functionalities such as web browsers, calculators, code interpreters and specialized APIs, agents can retrieve real-time data, perform complex computations or even execute software commands [LWZ+24]. The process of utilizing a tool involves several sub-tasks: intent recognition to determine when a tool is needed, function selection to choose the most appropriate tool, parameter-value mapping to extract required arguments and the actual execution and response generation [YEL+25].

Advanced agentic systems are not just limited to using pre-existing tools. These systems also have the ability to autonomously select, combine and even create new tools. Because language models can generate executable code, agents can write new Python utility functions or scripts on the fly, write their own unit tests and deploy them as custom tools to solve problems. Dynamic tool integration reduces computational errors and enhances the interpretability of an agent's actions. It also embeds operational rules in a clear, reliable manner [Sch25].

This was made possible by a major shift toward standardized protocols like the Model Context Protocol (MCP). MCP establishes a universal, framework-independent standard that decouples tool capabilities from the LLM, standardizing how agents securely connect to and invoke external computational resources [PBPS25, DBM25].

4.1.7 Guardrails and Governance

Because agentic AI operates with high autonomy in dynamic environments, robust guardrails are essential to ensure that systems act safely, predictably and in alignment with human values. Guardrails consist of embedded technical constraints, including goal safety protocols, schema validators and fail-safe mechanisms. They monitor and validate agent outputs before execution, preventing harmful actions or deviations from assigned

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

objectives. By incorporating node validation, retry logic and early-stage trust layers, developers can secure agent workflows and mitigate risks like malicious code execution or data privacy breaches [AKA25] [DBM25].

On a broader systemic level, governance frameworks dictate the degree of human oversight and accountability required for agentic systems. This is implemented through Human-in-the-Loop architectures, where a human must explicitly approve critical decisions, or Human-on-the-Loop designs, where humans monitor autonomous execution with the power to intervene with an override protocol. Governance also relies on Explainable AI techniques and comprehensive audit trails to log reasoning steps and tool invocations. This ensures that the agent's decision-making process remains transparent, legally compliant and accountable when errors occur [AKA25].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.1.8 Summary of Literature Study

The concept matrix in Table 4.1 documents the outcome of the literature study in Section 4.1. Each source in the literature corpus is mapped to the core concepts it addresses, thereby providing the evidence for the seven recurring concept areas. Each row corresponds to one source and each column to one of the seven concept areas, where a marker indicates that the source explicitly addresses that concept. Together, the seven concept areas form the conceptual baseline that this phase set out to produce. Because they come from the whole corpus, they do not depend on a single author's terminology. This baseline defines what a semantic representation of such systems needs to model, and it provides the criteria for the framework analysis in the following section, where each concept is examined against how CrewAI, AutoGen and LangGraph concretely implement it.

Table 4.1: Concept Matrix: Core Concepts across the Literature Corpus

Paper	Agent Def.	Percep./Task	Coord.	Reason.	Memory	Tools	Guard./Gov.
[WJ95]	•		•	•			
[HL04]			•				
[Woo09]	•	•	•	•			
[WBZ+23]	•	•	•	•	•	•	•
[HZC+24]	•	•	•	•		•	•
[GCW+24]	•	•	•	•	•	•	
[LWZ+24]	•	•	•	•	•	•	
[CLH+25]	•		•	•	•	•	
[Sch25]	•		•	•	•	•	
[YEL+25]				•	•	•	•
[Bot25]	•		•	•	•		
[DBM25]	•	•	•	•	•	•	•
[WZ25]	•	•	•	•	•	•	•
[AKA25]	•		•	•	•	•	•
[AAD25]							•
[PBPS25]	•	•	•			•	
[MRV+25]	•	•					•
[BBP25]	•	•	•				
[BKN+25]	•	•	•	•	•	•	•
[EKEK26]	•	•	•		•	•	
[SRK26]	•		•	•	•	•	
[VRB26]	•	•	•	•	•	•	•
[LFM+26]			•				•
[NSSP26]	•		•	•	•	•	

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.2 An Analysis of Agentic AI Frameworks

After establishing and analyzing the seven core concepts at the literature level, this section moves to the source code level and examines how each concept is realised across the three target frameworks (i.e. CrewAI [Cre25], AutoGen [Mic25] and LangGraph [Lan25]). For each of the seven concepts, the official documentation of each framework was searched for the mechanisms that implement it and the results were consolidated to show where the frameworks converge and where they diverge. From this consolidated comparison, the framework-specific mechanisms are transformed into a set of representational requirements that specify what a framework-independent semantic representation must capture. The requirements are consolidated in Section 4.3.

4.2.1 Agent Definition

CrewAI CrewAI defines an Agent as an autonomous unit capable of performing tasks, making decisions, using tools, interacting with other agents, maintaining memory of interactions and being able to delegate tasks when allowed. In the source code, agents are defined either using YAML configuration or directly in code. To define an agent's profile, CrewAI provides a list of agent attributes that can be specified when designing an agentic system. The only required parameters are role, goal and backstory [Cre25].

```yaml
data_analyst:
  role: Data Analyst
  goal: >
    Extract actionable insights from structured datasets
    and surface relevant patterns to the research team.
  backstory: >
    A seasoned analyst with expertise in statistical methods
    and business intelligence who prioritises clarity over
    complexity when communicating findings.
```

Listing 4.1: CrewAI agent definition (YAML configuration).

AutoGen AutoGen provides a set of preset agent types that all inherit from the BaseChatAgent class. All agents share a common set of attributes that define their identity: name, description and system_message, which are configured at agent instantiation. The name serves as a unique identifier within a multi-agent system. The description communicates the agent's purpose to other agents during orchestration and the system_message provides the behavioural instructions passed to the underlying language model [Mic25].

Among the preset agents, AssistantAgent serves as a general-purpose agent intended for prototyping and educational use. For more specialized requirements, AutoGen allows developers to define custom agents by subclassing BaseChatAgent, enabling the implementation of domain-specific behaviour that does not fit the provided presets [Mic25].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

LangGraph LangGraph, while part of the LangChain ecosystem, follows a different principle at the source code level than LangChain's agent abstractions. LangGraph explicitly distinguishes between workflows and agents: workflows follow predetermined code paths with a fixed execution order, whereas agents are driven by a language model that dynamically determines its own sequence of actions, typically through iterative tool use [Lan25].

LangGraph does not provide their users with a mechanism to configure specific agents with roles, descriptions, or backstories. Instead, behaviour is constructed using discrete steps called nodes, connected via edges that define transitions and conditional branching. An agent in LangGraph is therefore not defined through identity parameters but rather emerges as a node in which a language model autonomously decides whether to invoke tools or return a final response, typically implemented through a react-style loop [Lan25].

```
def data_analyst(state: MessagesState):
    system = SystemMessage(content=(
        "You are a data analyst with expertise in statistical "
        "methods and business intelligence. Prioritise clarity "
        "over complexity when communicating findings."
    ))
    response = model.invoke([system] + state["messages"])
    return {"messages": [response]}
```

Listing 4.2: LangGraph agent definition as a node function

Synthesis and Requirements The framework analysis of agent definitions revealed structural inconsistencies on an architectural level across all three frameworks. While CrewAI and AutoGen support defining individual agents with distinct characteristics and roles through dedicated parameters, LangGraph follows a different approach in which agents are not explicitly declared but emerge from graph nodes that contain language model calls with tool access.

Despite these differences, all three frameworks share the underlying concept of an autonomous unit that is associated with a language model and guided by natural language instructions. CrewAI expresses this through role, goal and backstory; AutoGen through name, description and system_message and LangGraph implicitly through prompt content within node functions.

From this analysis, the following representational requirements for agent definitions can be derived:

R1 Framework-independent agent identity. The ontology must represent agent identity as a separable concept that can be populated either from explicit configuration attributes (as in CrewAI and AutoGen) or inferred from prompt-level content within computational nodes (as in LangGraph).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

R2 Distinction between orchestration-relevant and model-relevant attributes. The ontology must distinguish between agent attributes that serve the orchestration layer (e.g. AutoGen's description used for speaker selection) and those that instruct the language model (e.g. AutoGen's system_message, CrewAI's backstory). A single agent may carry both types of directive simultaneously.

R3 Prompt decomposition into structured components The ontology must decompose natural language prompt content into structured components so that equivalent prompts can be reconstructed for different target frameworks from these components.

R4 Flexible attribute cardinality. The required and optional attributes differ across frameworks. CrewAI mandates role, goal and backstory; AutoGen requires only name; LangGraph has no required agent-level attributes at all. The ontology must define a minimal set of identity properties while accommodating framework-specific extensions without loss of information.

R5 Agent functional typing. The ontology must represent the functional role an agent plays within the coordination structure by distinguishing between general-purpose agents, manager/orchestrator agents and agents that serve as proxies for human input (independently of how this role is expressed in framework-specific class hierarchies).

4.2.2 Perception and Task Definition

CrewAI CrewAI lets developers define a Task as a specific assignment completed by an agent. Similar to the Agent, tasks can be defined as YAML configuration or directly in the code. On a source code level, a task is created with two required parameters: description, a natural language statement of what the task entails and expected_output, a description of what the completed task should produce. Tasks are optionally assigned to a specific agent via the agent parameter; if no agent is assigned, a hierarchical process can delegate tasks based on agent roles [Cre25].

Tasks can depend on other tasks through the context attribute, which accepts a list of tasks whose outputs should be available as input. CrewAI also supports task-level tool assignment, allowing a task to override the agent's default tools, as well as output formatting through structured types such as output_pydantic and output_json [Cre25].

AutoGen In AutoGen, there is no dedicated task class or configuration object. Instead, a task is expressed as a message passed to an agent or team through the run or run_stream methods via the task parameter, which accepts a plain string. The task description, expected output and any constraints must all be encoded within this single string. There is no built-in mechanism for defining task dependencies, structured output formats, or guardrails at the task level. Such logic must be implemented through agent orchestration or custom code [Mic25].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

LangGraph LangGraph does not have an explicit task concept. What other frameworks model as a task is instead encoded in the graph structure itself. A nodes function body contains the instructions and the graph's edges define the execution order and dependencies. Task descriptions, expected outputs and constraints are declared within node implementations rather than as separate, configurable objects [Lan25].

Synthesis and Requirements The framework analysis of task definitions revealed three distinct abstraction levels. CrewAI provides tasks as first-class, richly configurable objects with dedicated attributes for descriptions, expected outputs, agent assignment, dependencies, guardrails and structured output types. AutoGen reduces a task to a string passed at the systems runtime. There is no separation between the task description and its constraints. LangGraph does not have an explicit task concept. The logic is distributed across node functions and graph edges.

From this analysis, the following representational requirements can be derived:

R6 Task as a first-class ontological concept. The ontology must represent tasks as distinct entities with their own properties, even when the source framework does not model them explicitly. For AutoGen, this requires extracting task semantics from run method; for LangGraph, from node function bodies and associated prompts.

R7 Agent-task assignment. The ontology must represent the relationship between a task and the agent responsible for its execution. This includes explicit assignment (CrewAI's agent parameter), implicit assignment through orchestration (AutoGen's team routing) and structural assignment through graph topology (LangGraph's node-to-function binding).

R8 Data dependencies. The ontology must capture ordering and data dependencies between tasks. CrewAI models these explicitly through the context attribute, LangGraph encodes them as graph edges and AutoGen relies on sequential team execution or manual orchestration. The semantic representation must normalize these into a unified dependency model.

R9 Expected output. The ontology must represent the expected output of a task, both as a natural language description and, where available, as a structured schema definition.

4.2.3 Coordination

CrewAI CrewAI implements coordination at two distinct levels: *Crews* for organizing agents around a shared set of tasks and *Flows* for orchestrating multiple crews and coding tasks into larger workflows. A Crew is a team of agents assigned to a list of tasks, governed by a *process* that determines execution order. CrewAI provides two process types: sequential, where tasks are executed in the order they are defined with the output of one task serving as context for the next and hierarchical, where a manager agent

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

dynamically delegates tasks to other agents based on their roles and capabilities. The process type is set at crew creation via the process parameter. In hierarchical mode, either a manager_llm or a custom manager_agent must be specified [Cre25].

Flows operate at a higher abstraction level. They connect multiple crews, individual agents and Python functions into event-driven workflows. A Flow is defined as a Python class inheriting from Flow, with methods decorated as entry points (@start), listeners (@listen), or routers (@router). Listeners are triggered when a specified method completes, which enables sequential chaining. Conditional logic is supported through or_ (trigger when any dependency completes), and_ (trigger when all dependencies complete) and router methods that return string labels to direct execution to different branches. Flows maintain their own state, which can be either unstructured (dictionary) or structured (Pydantic model) and is accessible across all methods in the flow [Cre25].

AutoGen AutoGen implements coordination through *teams*, which are groups of agents that share a conversation context and take turns responding according to a defined interaction pattern. A team is instantiated by passing a list of agents and a termination condition that determines when execution stops. AutoGen provides several team presets, each implementing a different speaker selection strategy:

- RoundRobinGroupChat: Agents take turns in a fixed round-robin order. Each agent broadcasts its response to all other agents, maintaining a shared context.
- SelectorGroupChat: A language model selects the next speaker after each message.
- MagenticOneGroupChat: A generalist multi-agent configuration designed for solving open-ended web and file-based tasks.
- Swarm: Agents use HandoffMessage to explicitly signal transitions to specific other agents.

```
termination = (
    TextMentionTermination("DONE")
    | MaxMessageTermination(max_messages=6)
)

team = RoundRobinGroupChat(
    participants=[researcher, critic],
    termination_condition=termination,
)
```

Listing 4.3: AutoGen RoundRobinGroupChat with combined termination conditions.

Termination conditions control when a team stops executing. These are passed at team creation and include conditions such as TextMentionTermination (stops when a specific word appears), TextMessageTermination (stops when a specific agent produces a text

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

message) and ExternalTermination (stops from outside the team). Conditions can be combined using bitwise operators [Mic25].

LangGraph LangGraph encodes all coordination logic explicitly through a directed graph composed of nodes and edges. There is no dedicated team, crew, or orchestration abstraction. The graph itself is the coordination mechanism. Nodes represent discrete computational steps (agent calls, tool execution, routing logic) and edges define the transitions between them. This gives developers full control over the execution flow but requires all coordination patterns to be manually constructed [Lan25].

On a source code level, a graph is built using the StateGraph class, which is parameterized with a typed state schema. Nodes are added via add_node and transitions are defined through add_edge for unconditional transitions or add_conditional_edges for branching based on the output of a node. Conditional edges accept a routing function that inspects the current state and returns the name of the next node to execute [Lan25].

LangGraph provides prebuilt utilities that implement common coordination patterns. The create_react_agent function constructs a ReAct-style loop where an agent node and a tool node alternate until the agent produces a final response. For multi-agent coordination, developers manually compose multiple agent nodes within a single graph and define the routing logic between them [Lan25].

Synthesis and Requirements The framework analysis of coordination revealed three different paradigms. Despite these differences, all three frameworks ultimately express the same fundamental coordination concerns: which agents participate, in what order they act and under what conditions execution branches or terminates. The difference is whether these concerns are declared through configuration (CrewAI's process type), selected from presets (AutoGen's team types), or constructed from primitives (LangGraph's nodes and edges).

From this analysis, the following representational requirements can be derived:

R10 Coordination pattern as a named concept. The ontology must represent coordination patterns (sequential, hierarchical, round-robin, swarm, selector-based) as identifiable concepts, regardless of whether they are selected from a preset (CrewAI, AutoGen) or manually constructed in a graph (LangGraph). This enables cross-framework comparison at the architectural level.

R11 Agent-to-coordination binding. The ontology must capture which agents participate in a coordination structure and how they are bound to it. Either as members of a crew with assigned tasks (CrewAI), as participants in a team (AutoGen), or as nodes within a graph (LangGraph).

R12 Termination conditions. All three frameworks define conditions under which execution stops, though the mechanisms differ: CrewAI relies on task completion and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

flow structure, AutoGen uses explicit termination condition objects and LangGraph terminates when a node routes to the END sentinel. The ontology must capture termination semantics independently of these framework-specific mechanisms.

R13 Multi-level coordination. CrewAI's separation of crews (inner coordination) and flows (outer orchestration) has no direct equivalent in AutoGen or LangGraph, where a single abstraction handles both levels. The ontology must be able to represent nested coordination structures so that a CrewAI flow containing multiple crews can be captured without flattening the hierarchy, while also representing the single-level coordination of the other frameworks.

R14 Workflow step structure with typed transitions. The ontology must represent the execution structure of a coordination pattern as a sequence of discrete steps with typed transitions. This includes distinguishing entry points, exit points and intermediate steps; representing unconditional and conditional transitions with associated routing logic; and linking each step to its associated task where applicable.

4.2.4 Reasoning

CrewAI CrewAI provides reasoning as an explicit, configurable feature at the agent level. On a source code level, this is achieved by setting a boolean flag reasoning in the agent definition. When set to True, the agent reflects on a given task, evaluates possible approaches and refines its plan before executing an action. This reflection cycle repeats until the agent converges on a satisfactory plan or a configurable max_reasoning_attempts threshold, defined as an integer in the agent definition, is reached. Reasoning in CrewAI is therefore a framework-managed capability that is toggled per agent independently of the underlying language model [Cre25].

AutoGen In AutoGen, reasoning is not exposed as a dedicated framework feature but is integrated into its conversation-based architecture. Whether and how an agent reasons depends primarily on the capabilities of the underlying language model. If a reasoning-capable model such as o3-mini is selected as the model client, the model itself handles multi-step reasoning internally without additional framework configuration. Alternatively, reasoning can be achieved architecturally by introducing a dedicated planning agent that decomposes a task into sub-steps using a language model before delegating them to other agents. In this case, reasoning emerges from the orchestration of multiple agents rather than from a single agent's configuration [Mic25].

LangGraph LangGraph does not provide a built-in reasoning mechanism at the agent or node level. Since LangGraph models agentic behaviour as explicit graph structures, reasoning patterns must be manually constructed by the developer through the graph topology itself. A Chain of Thought pattern, for instance, can be implemented as a sequence of nodes where each node performs one reasoning step and passes its output via

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the shared state to the next. More complex patterns such as reflection or self-correction require the developer to define conditional edges that loop back to previous nodes based on evaluation criteria. LangGraph therefore offers maximum flexibility in designing reasoning workflows but places the full architectural decision making on the developer, as no reasoning behaviour is provided by default [Lan25].

Synthesis and Requirements The framework analysis of reasoning capabilities revealed three different abstraction levels. CrewAI treats reasoning as a framework-managed, declarative feature that is toggled per agent through a boolean flag and bounded by a configurable iteration limit. AutoGen delegates reasoning either to the underlying language model's native capabilities or to the architectural composition of multiple agents, where a dedicated planning agent orchestrates task decomposition. LangGraph provides no built-in reasoning mechanism and instead requires developers to manually encode reasoning patterns through the graph topology itself.

From this analysis, the following representational requirement for reasoning can be derived:

R15 Reasoning as capability, pattern and origin. The ontology must represent reasoning on three dimensions: (1) whether an agent has reasoning enabled as a declarative capability, (2) which reasoning pattern is employed if identifiable and (3) whether reasoning originates from the framework, the language model's native capabilities, or the system's graph topology.

4.2.5 Memory

CrewAI CrewAI provides an integrated Memory class that supports short-term, long-term, entity and external memory types. This class uses a language model to analyze content at storage time, inferring scope, categories and importance. The memory system can be used as a standalone component, at the agent level, at the crew level and within flows [Cre25].

At the crew level, all agents in a crew share the crew's memory unless an individual agent has its own. After each task, the crew automatically extracts facts from the task output and stores them. Before the next task, relevant context is recalled from memory and injected into the task prompt at runtime. At the agent level, agents can maintain private memory that is not shared with the rest of the crew. Standalone memory acts as a general-purpose knowledge base that is independent of any specific agent or crew [Cre25].

AutoGen AutoGen provides a Memory protocol that defines a standard interface for storing and retrieving contextual information. The protocol specifies methods for adding entries, querying relevant content, updating an agent's model context and clearing the store. The simplest implementation is ListMemory, which maintains memory entries in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

chronological order and appends them to the model's context before each task execution [Mic25].

At source code level, memory is assigned to an agent by passing a list of memory instances during agent instantiation via the memory parameter. At runtime, the agent queries its memory stores before processing a task. The retrieved content is added into the model context as a system message. Memory in AutoGen is therefore agent-scoped. This means each agent manages its own memory instances independently [Mic25].

For more advanced use-cases, AutoGen provides memory extensions such as ChromaDBVectorMemory and RedisMemory, which implement the same Memory protocol but use vector similarity search for retrieval rather than chronological ordering. Since all implementations conform to the same protocol, they are interchangeable at the source code level without modifying the agent's logic [Mic25].

LangGraph LangGraph implements memory through two distinct mechanisms that map to different persistence scopes. Short-term memory is provided through *checkpointers*, which persist the graph's state across invocations within the same thread. On a source code level, a checkpointer is instantiated and passed to the graph at compile time via the checkpointer parameter. Every invocation carries a thread_id and the checkpointer automatically saves and restores the full graph state between turns. [Lan25].

Long-term memory is implemented through a separate *store* abstraction, which persists data across threads and sessions. A store is similarly passed to the graph at compile time via the store parameter. Unlike the checkpointer, which operates transparently on the entire graph state, the store must be explicitly accessed within node functions to read and write memory entries. Entries are organised into namespaces, typically scoped by user or application context [Lan25].

Synthesis and Requirements CrewAI provides a unified memory system that operates at both crew and agent level, with automatic fact extraction and injection at runtime. AutoGen exposes memory through a protocol-based interface that is strictly agent-scoped, with no built-in shared memory across agents. LangGraph separates memory into two distinct mechanisms (i.e. checkpointers for short-term state persistence and stores for long-term data) both scoped to the graph as a whole and accessed explicitly within node functions.

From this analysis, the following representational requirements can be derived:

R16 Memory scope as an explicit property. The ontology must represent at which level memory is owned and accessible: agent-private, shared among a group, or global to the system.

R17 Separation of short-term and long-term persistence. The ontology must represent the lifecycle of memory, whether it persists within a single execution,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

across executions within a thread, or across all sessions independently of how a given framework packages its memory features.

4.2.6 Tool Integration

CrewAI CrewAI defines a tool as a skill or function that agents can utilise to perform actions. Tools can be assigned at two levels: at the agent level, where they become available across all tasks the agent executes and at the task level, where they override the agent's default tool set for that specific task [Cre25].

On a source code level, custom tools are created either by subclassing BaseTool or by using the @tool decorator on a function. When subclassing, the developer defines a name, a description, an input schema via args_schema (a Pydantic model) and implements the _run method containing the tool's logic. When using the decorator, the function's docstring serves as the tool description that the agent uses to decide when to invoke it. CrewAI also provides a large library of pre-built tools (e.g. SerperDevTool, FileReadTool, PDFSearchTool) that can be instantiated and assigned to agents or tasks without custom implementation [Cre25].

AutoGen AutoGen enables tool use through the AssistantAgent, which accepts tools via the tools parameter. A tool is most commonly defined as a Python function, which the agent automatically converts into a FunctionTool at runtime. The tool schema (including name, description and parameter types) is automatically generated from the function's signature and docstring and provided to the language model during execution [Mic25].

Beyond individual tools, AutoGen introduces the concept of a *Workbench*, which is a collection of tools that share state and resources. The primary example is McpWorkbench, which connects to a Model Context Protocol (MCP) server and exposes its tools to the agent. A workbench is passed to the agent via the workbench parameter and operates alongside any individually assigned tools [Mic25].

LangGraph LangGraph implements tool use through LangChain's tool infrastructure. Tools are defined as Python functions using the @tool decorator, where the function's docstring serves as the description and type hints define the input schema. For more complex inputs, a Pydantic model or JSON schema can be specified via the args_schema parameter [Lan25].

Tools are integrated into the graph through a prebuilt ToolNode, which handles tool execution, parallel tool calls and error handling. The ToolNode is added to the graph like any other node and connected to the model node via a conditional edge (tools_condition) that routes execution to the tool node only when the model generates a tool call [Lan25].

Synthesis and Requirements All three frameworks share the concept of a tool as a callable function with a name, description and typed input schema that is exposed to a language model. This shows that all frameworks support defining tools as decorated

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Python functions with auto-generated schemas. The key differences lie in how tools are assigned, scoped and integrated into the execution flow.

From this analysis, the following representational requirements can be derived:

R18 Tool as a framework-independent concept. The ontology must represent tools as entities with a name, description and input schema, independent of the framework-specific definition mechanism (decorator, subclass, or function).

R19 Tool assignment scope. The ontology must capture at which level a tool is bound: to an individual agent (all frameworks), to a specific task (CrewAI), or to a graph node (LangGraph).

4.2.7 Guardrails and Governance

CrewAI CrewAI implements guardrails at the task level through the guardrail and guardrails parameters. First, function-based guardrails, which are Python functions that receive a TaskOutput and return a tuple indicating success or failure with feedback. Second, LLM-based guardrails, which are string descriptions that the agent's language model evaluates against the output. Multiple guardrails can be chained sequentially, with each receiving the output of the previous one. A configurable guardrail_max_retries parameter limits how many times an agent reattempts a task after guardrail failure [Cre25].

AutoGen AutoGen does not provide a dedicated guardrail mechanism at the task or agent level. Output validation must be implemented by the developer, typically through a dedicated critic agent that evaluates outputs and provides feedback within the team's conversational flow. A validator agent is included as a team participant and a TextMentionTermination condition (e.g. listening for "APPROVE") controls when execution stops [Mic25].

Human-in-the-loop is supported through two complementary mechanisms. First, a UserProxyAgent can be included as a participant in a team, prompting the user for input during its turn via a configurable input function. This approach is synchronous and blocks the team's execution until the user responds. Second, the team can be configured to pause after a set number of agent turns using the max_turns parameter, or to stop when an agent issues a HandoffMessage targeting the user, detected by a HandoffTermination condition. In these cases, the team preserves its state after stopping and can be resumed with user feedback passed as the task parameter in the next call to run [Mic25].

LangGraph LangGraph does not provide built-in guardrails but offers comprehensive mechanisms for human-in-the-loop through its interrupt function. Unlike static breakpoints (which pause before or after specific nodes at compile time), interrupts are dynamic. They can be placed anywhere within a node's logic and can be conditional based on application state. Calling interrupt suspends graph execution and the current

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

state is saved via the checkpointer. Additionally, the payload is shown to the caller. Execution continues when the caller re-invokes the graph with a Command(resume=...) containing the human's response. This becomes the return value of the interrupt call inside the node [Lan25].

This mechanism supports several human-in-the-loop patterns at the source code level. Approval workflows where a node pauses for a binary approve/reject decision, review-and-edit patterns where a human can modify LLM outputs before they are committed to state and input validation loops where an interrupt repeatedly prompts the user until valid input is received. Interrupts can also be placed inside tool functions, causing the tool to pause for approval whenever it is invoked. Guardrail-like validation must be implemented as a dedicated node in the graph that inspects the state and routes execution to either the next step or back to the producing node via conditional edges [Lan25].

Synthesis and Requirements The frameworks diverge significantly in how they handle validation and human oversight. CrewAI provides guardrails as a first-class, configurable feature at the task level with built-in retry logic and supports human-in-the-loop through task-level flags and flow-level decorators. AutoGen has no dedicated guardrail mechanism and validation is achieved through critic agents participating in the conversational flow, while human-in-the-loop is supported through UserProxyAgent participation, max_turns pausing and HandoffTermination conditions. LangGraph requires guardrail logic to be encoded as validation nodes with conditional edges and provides a dynamic interrupt function that can be placed anywhere within node logic for human review, approval, or input validation.

From this analysis, the following representational requirements are derived:

R20 Guardrails as a task-level property. The ontology must represent that a task or step has associated validation logic, regardless of whether it is implemented as a task parameter (CrewAI), a validator agent (AutoGen), or a conditional node (LangGraph). The representation captures that validation exists and what criteria it checks, without modelling the implementation mechanism itself.

R21 Human-in-the-loop as an explicit checkpoint. The ontology must represent points in the workflow where human review or input is required. This includes CrewAI's human_input flag and @human_feedback decorator, AutoGen's UserProxyAgent and HandoffTermination targeting the user and LangGraph's interrupt calls within nodes. The semantic representation captures the presence and position of human checkpoints so that the transformation method can reconstruct an equivalent mechanism in the target framework.

4.3 Results of Requirements Elicitation

From the synthesis of each concept area, a total of 21 representational requirements were derived. Each requirement specifies what the semantic representation must capture to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

enable framework-independent modelling of the corresponding concept. Together, the seven core concepts and the 21 requirements define the scope and acceptance criteria for the ontology engineering phase that follows.

Table 4.2: Representational Requirements for a Semantic Model of Agentic AI Systems

ID	Concept	Requirement	Relevant Literature
R1	Agent Def.	**Framework-independent agent identity.** The ontology must represent agent identity as a separable concept that can be populated either from explicit configuration attributes (as in CrewAI and AutoGen) or inferred from prompt-level content within computational nodes (as in LangGraph).	[Woo09, EKEK26, LWZ+24, GCW+24, Mic25, Lan25, Cre25]
R2	Agent Def.	**Distinction between orchestration-relevant and model-relevant attributes.** The ontology must distinguish between agent attributes that serve the orchestration layer and those that instruct the language model. A single agent may carry both types of directive simultaneously.	[DBM25, WBZ+23, Mic25, BBP25, VRB26]
R3	Agent Def.	**Prompt decomposition into structured components.** The ontology must decompose natural language prompt content into structured components so that equivalent prompts can be reconstructed for different target frameworks from these components.	[GCW+24, LWZ+24, EKEK26, Cre25, Mic25, Lan25]
R4	Agent Def.	**Flexible attribute cardinality.** The ontology must define a minimal set of identity properties while accommodating framework-specific extensions without loss of information.	[DBM25, EKEK26, Cre25, Mic25, Lan25]
R5	Agent Def.	**Agent functional typing.** The ontology must represent the functional role an agent plays within the coordination structure by distinguishing between general-purpose agents, manager/orchestrator agents and agents that serve as proxies for human input.	[SRK26, WBZ+23, BBP25, Cre25, Mic25]
R6	Perception / Task	**Task as a first-class ontological concept.** The ontology must represent tasks as distinct entities with their own properties, even when the source framework does not model them explicitly.	[HZC+24, VRB26, BKN+25, EKEK26, Cre25]
R7	Perception / Task	**Agent-Task assignment.** The ontology must represent the relationship between a task and the agent responsible for its execution, including explicit assignment, implicit assignment through orchestration and structural assignment through graph topology.	[EKEK26, BBP25, VRB26, Cre25, Lan25]
R8	Perception / Task	**Data dependencies.** The ontology must capture ordering and data dependencies between tasks and normalize these into a unified dependency model.	[EKEK26, DBM25, GCW+24, HZC+24, Cre25, Lan25]
R9	Perception / Task	**Expected outputs.** The ontology must represent the expected output of a task, both as a natural language description and, where available, as a structured schema definition.	[HZC+24, VRB26, EKEK26, DBM25, WBZ+23, Cre25, Lan25, Mic25]
R10	Coordination	**Coordination pattern as a named concept.** The ontology must represent coordination patterns (sequential, hierarchical, round-robin, swarm, selector-based) as identifiable concepts regardless of whether they are selected from a preset or manually constructed in a graph.	[GCW+24, VRB26, CLH+25, Cre25, Mic25, BBP25, Lan25]
R11	Coordination	**Agent-to-coordination binding.** The ontology must capture which agents participate in a coordination structure and how they are bound to it.	[LWZ+24, WBZ+23, Cre25, Mic25, Lan25]
R12	Coordination	**Termination conditions.** The ontology must capture termination semantics independently of framework-specific mechanisms.	[WBZ+23, BKN+25, Cre25, Mic25, Lan25]
R13	Coordination	**Multi-level coordination.** The ontology must represent nested coordination structures so that a CrewAI flow containing multiple crews can be captured without flattening the hierarchy, while also representing single-level coordination.	[BKN+25, LWZ+24, EKEK26, LFM+26, Cre25]

Continued on next page

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

ID	Concept	Requirement	Relevant Literature
R14	Coordination	**Workflow step structure with typed transitions.** The ontology must represent the execution structure as a sequence of discrete steps with typed transitions, distinguishing entry points, exit points, intermediate steps, unconditional and conditional transitions with associated routing logic.	[EKEK26, VRB26, BBP25, Cre25, Lan25]
R15	Reasoning	**Reasoning as capability, pattern and origin.** The ontology must represent whether an agent has reasoning enabled, which reasoning pattern is employed if identifiable and whether reasoning originates from the framework, the model or the system topology.	[Sch25, BBP25, NSSP26, YEL+25, Cre25, Mic25, Lan25]
R16	Memory	**Memory scope as an explicit property.** The ontology must represent at which level memory is owned and accessible: agent-private, shared among a group or global to the system.	[EKEK26, SRK26, DBM25, Cre25, Mic25, Lan25]
R17	Memory	**Separation of short-term and long-term persistence.** The ontology must represent the lifecycle of memory independently of how a given framework packages its memory features.	[DBM25, SRK26, Sch25, BKN+25, Cre25, Lan25, Mic25]
R18	Tools	**Tool as a framework-independent concept.** The ontology must represent tools as entities with a name, description and input schema, independent of the framework-specific definition mechanism.	[DBM25, PBPS25, EKEK26, Cre25, Mic25, Lan25]
R19	Tools	**Tool assignment scope.** The ontology must capture at which level a tool is bound: to an individual agent, to a specific task or to a graph node.	[VRB26, DBM25, EKEK26, Mic25, Lan25, Cre25]
R20	Guardrails	**Guardrails as a task-level property.** The ontology must represent that a task has associated validation logic, regardless of whether it is implemented as a task parameter, a validator agent or a conditional node.	[BKN+25, DBM25, VRB26, Cre25, Mic25, Lan25]
R21	Guardrails	**Human-in-the-loop as an explicit checkpoint.** The ontology must represent points in the workflow where human review or input is required, capturing the presence and position of human checkpoints.	[AKA25, AAD25, VRB26, WBZ+23, Cre25, Lan25, Mic25]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

AgentOSCIN: An Ontology for Agentic AI Framework Interoperability

The previous chapter analyzed how agentic AI concepts are realised across modern Agentic AI frameworks and derived a set of representational requirements from those findings. These requirements describe what a framework-independent semantic representation must be able to express in order to support interoperability. This chapter presents the ontology developed to meet those requirements. The ontology is the first of the two artefacts of this thesis. It serves as the common intermediate representation and needs to be expressive enough for framework-agnostic code extraction and generation.

5.1 Scope and Competency Questions

The scope is defined through competency questions, as introduced by Gruninger and Fox [GF95]. They describe competency questions as natural language queries that a fully populated ontology must be able to answer. If a populated ontology instance cannot answer a competency question, either the ontology is missing a required concept or the extraction has not captured it [NM01]. In this thesis, the competency questions (see Tab. 5.1) are derived directly from the requirements established in Chapter 4.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Table 5.1: CQs derived from Requirements Elicitation

CQ	Req.	Concept	Competency Question
CQ-1	R1	Agent Def.	Which agents are defined in this system?
CQ-2	R1	Agent Def.	What identifier and role is assigned to each agent?
CQ-3	R1	Agent Def.	What category (worker / manager / human-proxy) does the agent declare?
CQ-4	R2	Agent Def.	Does the agent distinguish orchestration-layer from language-model prompts?
CQ-5	R2	Agent Def.	What explicit directive role does each prompt declare?
CQ-6	R3	Agent Def.	What are the instruction, context and output indicator components of a given agent's prompt?
CQ-7	R4	Agent Def.	Which core agent properties (id, role, language model, prompt, goal) are populated?
CQ-8	R4	Agent Def.	Which extension properties (type, reasoning pattern, memory binding) are populated?
CQ-9	R5	Agent Def.	Is a given agent a proxy for human input?
CQ-10	R5	Agent Def.	Is a given agent classified as worker or manager?
CQ-11	R6	Perception	What tasks does this system define and what does each task instruct the assigned agent to do?
CQ-12	R7	Perception	Which agent is responsible for a given task?
CQ-13	R7	Perception	How was that assignment determined (delegation strategy)?
CQ-14	R8	Perception	Which tasks must complete before a given task can start?
CQ-15	R8	Perception	What kind of dependency holds between a task and its predecessor (ordering only, or output-passing)?
CQ-16	R9	Perception	What output is a task expected to produce?
CQ-17	R9	Perception	Is a structured output schema defined for the task?
CQ-18	R10	Coordination	What coordination pattern does a given team follow?
CQ-19	R11	Coordination	Which agents belong to a given team?
CQ-20	R11	Coordination	How are those agents bound to the team's coordination structure?
CQ-21	R12	Coordination	Under what conditions does a team's execution stop?
CQ-22	R13	Coordination	Does a team contain sub-teams and if so how are they nested?
CQ-23	R14	Coordination	What is the step-by-step execution structure of a team's workflow?
CQ-24	R14	Coordination	Where does the workflow branch conditionally?
CQ-25	R15	Reasoning	Does the agent use reasoning at all?
CQ-26	R15	Reasoning	What reasoning pattern does it follow?
CQ-27	R15	Reasoning	Where does that reasoning capability originate?
CQ-28	R16	Memory	Is an agent's memory private, shared with the team or accessible system-wide?
CQ-29	R17	Memory	Does a given memory persist only during execution, across threads or across all sessions?
CQ-30	R18	Tools	What tools are available in this system?
CQ-31	R18	Tools	What input does each tool expect?
CQ-32	R19	Tools	Is a given tool bound at the agent level?
CQ-33	R19	Tools	Is a tool bound or overridden for a specific task?
CQ-34	R20	Guardrails	Does a task have validation guardrails and what type of check do they perform?
CQ-35	R21	Guardrails	At which points in the workflow must a human provide input or approval?

5.2 Ontology Reuse

The ontology is not built from scratch. AgentO [EKEK26] provides the conceptual foundation, defining core classes such as `LLMAgent`, `Task`, `Tool`, `Prompt` and `WorkflowStep`. Where AgentO's existing vocabulary satisfies a requirement, it is reused without modification. Where it falls short, new classes and properties are introduced. All extensions are strictly additive. This means that no existing AgentO class is removed or structurally modified and no existing property has its domain or range altered. Other concepts and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

properties derived from literature review in chapter 4, while not defined as ontology classes and properties, were also considered in reuse evaluation (e.g. the unified class model proposed by Derouiche et al. [DBM25])

To make the reuse explicit, Table 5.2 maps every AgentOSCIN class to its origin in AgentO [EKEK26] and to the corresponding concept in the unified class model of Derouiche et al.[DBM25], referred to as the DBM model. As the DBM model is a class-level sketch, the last column records a direct correspondence (✓), an informal description in prose only (∼) or no correspondence (×).

Table 5.2: Coverage of AgentOSCIN classes by AgentO [EKEK26] and the DBM model [DBM25] (✓ fully covered, ∼ partially covered, × not covered).

AgentOSCIN class	**AgentO**	**DBM**
Agent definition		
`:LLMAgent`	✓	✓
`:HumanAgent`	✓	∼
`:Prompt`	✓	✓
Perception and task		
`:Task`	✓	✓
`:Schema`	×	×
Coordination		
`:AgenticSystem`	∼	×
`:Team`	✓	∼
`:Orchestration`	∼	×
`:CoordinationPattern`	×	×
`:WorkflowPattern`	✓	×
`:WorkflowStep`	✓	×
`:ConditionalStep`	∼	×
`:TerminationCondition`	×	×
Reasoning		
`:ReasoningPattern`	×	∼
Memory		
`:Memory`	✓	✓
`:MemoryBinding`	×	×
Tools		
`:Tool`	✓	✓
Guardrails and governance		
`:Guardrail`	×	✓
`:HumanCheckpoint`	×	∼
✓ fully covered	**9**	**6**
∼ partially covered	**3**	**4**
× not covered	**7**	**9**

The remaining AgentOSCIN classes are structural superclasses reused unchanged from AgentO and its imported vocabularies: `:Agent` (prov), `:System`, `:Resource`, `:Instance`, `:Model` and `:StatisticalModel` (beam), `:Plan` and `:Step` (p-plan), as well as `:Capability`, `:Config`, `:Constraint`, `:Context`, `:Environment`,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

`:Goal`, `:KnowledgeBase`, `:LanguageModel`, `:Objective`, `:StartStep` and `:EndStep` (AgentO).

Table 5.3 lists the main properties that realise the seven concept groups, both those reused from AgentO [EKEK26] and those introduced by AgentOSCIN and records whether an equivalent property or relation is fully covered, partially covered or not covered by AgentO and by the DBM model. A property reused from AgentO is fully covered by AgentO by construction. Although the DBM model is presented as a class diagram, its associations act as relations comparable to object properties, for example `reason-via`, `use`, `assign`, `depend` and `validate`. Generic AgentO properties for configuration, capabilities, goals, resources and the environment are reused without modification.

Table 5.3: Coverage of the core AgentOSCIN properties by AgentO [EKEK26] and by the relations of the DBM model [DBM25] (✓ fully covered, ∼ partially covered, × not covered)

AgentOSCIN property	**AgentO**	**DBM**
Agent definition		
`:hasPrompt`	✓	✓
`:agentPrompt`	✓	✓
`:useLanguageModel`	✓	✓
`:agentID`	✓	×
`:agentRole`	✓	×
`:promptContext`, `:promptInstruction`, `:promptInputData`, `:promptOutputIndicator`	✓	×
`:agentType`	∼	×
`:hasDirectiveFunction`	∼	×
`:hasSourceAttribute`	×	×
Perception and task		
`:performedByAgent`	✓	✓
`:taskPrompt`	✓	×
`:hasAssociatedTask`	✓	×
`:dependsOn`	×	✓
`:hasDependencyType`	×	∼
`:hasDelegationStrategy`	×	∼
`:hasOutputSchema`	×	×
`:hasExpectedOutput`	×	×
`:hasSchemaDefinition`	×	×
Coordination		
`:interactsWith`	✓	✓
`:hasAgentMember`	✓	∼
`:hasWorkflowPattern`	✓	×
`:hasWorkflowStep`	✓	×
`:nextStep`	✓	×
`:relatedStep`	✓	×
`:nextPattern`	✓	×
`:hasRelatedPattern`	✓	×
`:hasSubPattern`	✓	×
`:stepOrder`	✓	×
`:containsAgent`	∼	∼
`:containsTeam`	×	×
`:containsOrchestration`	×	×
`:orchestratesTeam`	×	×

continued on next page

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

AgentOSCIN property	AgentO	DBM
:employsCoordinationPattern	×	∼
:hasTerminationCondition	×	×
:hasSubCondition	×	×
:hasSourceFramework	×	×
:hasRoutingLogic	×	×
:hasDecoratorArgument	×	×
:hasMaxTurns	×	×
:hasTriggerExpression	×	×
:hasOperator	×	×
Reasoning		
:employsReasoningPattern	×	∼
:hasReasoningEnabled	×	×
:hasReasoningOrigin	×	×
:hasMaxReasoningAttempts	×	×
Memory		
:hasKnowledge	✓	×
:hasMemoryBinding	×	∼
:hasAgentMemoryBinding	×	∼
:hasTeamMemoryBinding	×	×
:bindsMemory	×	×
:hasMemoryScope	×	×
:hasPersistenceScope	×	∼
Tools		
:agentToolUsage	✓	✓
:toolUsage	✓	✓
:taskToolUsage	∼	✓
:hasInputSchema	×	×
:hasImplementationReference	×	×
Guardrails and governance		
:humanParticipatedIn	✓	×
:hasGuardrail	×	✓
:hasValidationLogic	×	∼
:hasHumanCheckpoint	×	×
:hasGuardrailType	×	×
:hasMaxRetries	×	×
:hasCheckpointType	×	×
:hasCheckpointPosition	×	×
:isMandatory	×	×
✓ fully covered	**23**	**10**
∼ partially covered	**4**	**10**
× not covered	**39**	**46**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

5.3 Defining Classes and Properties

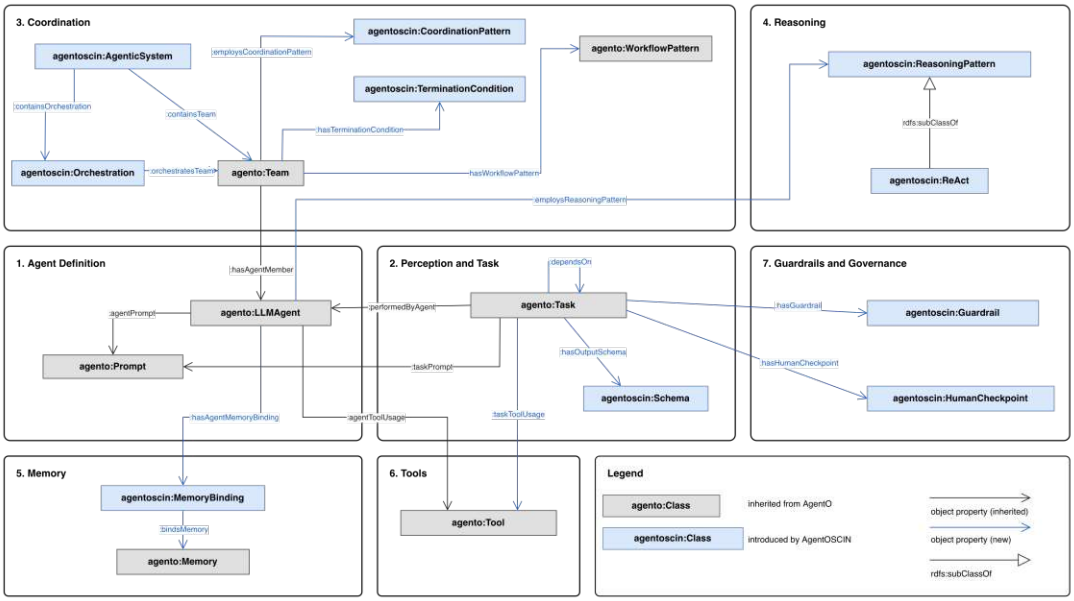

Figure 5.1: AgentOSCIN Overview: Core classes and properties divided into the concept groups

We propose AgentOSCIN as a framework-independent intermediate representation for agentic AI systems. In alignment with the requirements introduced in Chapter 4, the ontology is structured into seven interrelated concept groups: agent definition, perception and task, coordination, reasoning, memory, tools as well as guardrails and governance. These groups and their interconnections are presented in Figure 5.1 and described in Sections 5.3.1 to 5.3.7.

In the following part, we will go into more detail for each of the concept groups. Furthermore, we provide an example instantiation and a respective visual representation of each group based on a running example. The running example is a CrewAI code-review system, in which a code-review agent, a security-audit agent and a review-summarizer agent collaborate as a sequential crew that is wrapped in an orchestration flow. The prefixes defined in Listing 5.1 are used for all Turtle serialisations in this chapter. AgentOSCIN imports AgentO and reuses its terms under the `agento:` prefix, while terms introduced by AgentOSCIN use the default `:` prefix.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
@prefix :       <http://www.semanticweb.org/danilippmann/ontologies/2026/3/agentoscin/> .
@prefix agento: <http://www.w3id.org/agentic-ai/onto#> .
@prefix beam:   <http://w3id.org/beam/core#> .
@prefix ex:     <http://example.org/agentoscin/runningexample#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
```

Listing 5.1: Definition of prefixes used in this chapter.

5.3.1 Agent Definition

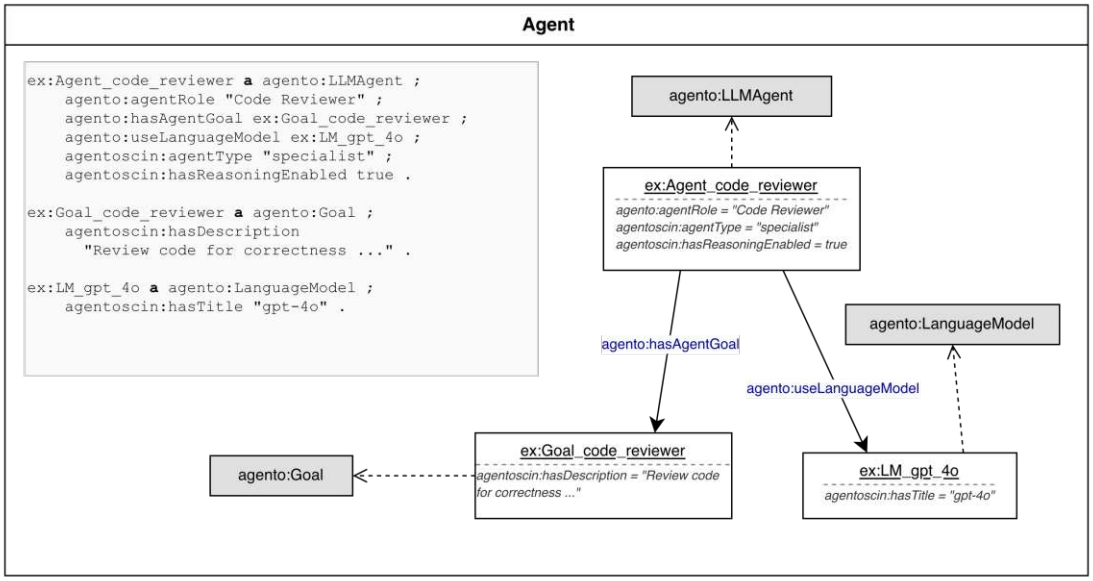

Figure 5.2: Agent definition instantiation excerpt.

The agent definition group captures the framework-independent identity of an agent through the `agento:LLMAgent` class and the `agento:Prompt` class it carries. Both are inherited from AgentO and together cover R1 (framework-independent agent identity) and R4 (flexible attribute cardinality) without modification. AgentOSCIN extends this baseline with properties that carry the remaining agent-level requirements.

The datatype property `:agentType` (R5) types each agent by its functional role in the coordination structure. This concept is modeled as a property and not as a subclass hierarchy, reflecting the cross-framework variability of the role: a manager agent in CrewAI is structurally different from a `UserProxyAgent` in AutoGen and both differ from a routing node in LangGraph [Cre25, Mic25, Lan25]. A property-based encoding

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

allows the same `agento:LLMAgent` individual to be classified consistently regardless of how its host framework expresses the role.

R2 (orchestration- versus model-relevant directives) and R3 (prompt decomposition) are handled at the `agento:Prompt` level rather than at the agent level. The directive function of a prompt fragment is an attribute of the prompt itself, not of the agent that carries it and decomposing a prompt into structured components only makes sense once the prompt is treated as a first-class entity. AgentO's existing `agento:Prompt` class is therefore retained as the basis for two new datatype properties. `:hasDirectiveFunction` records the function a prompt fragment serves, for example role instruction, context or backstory. `:hasSourceAttribute` preserves the name of the source code attribute the prompt was extracted from, for example `goal`, `backstory` or `system_message`.

5.3.2 Perception and Task Definition

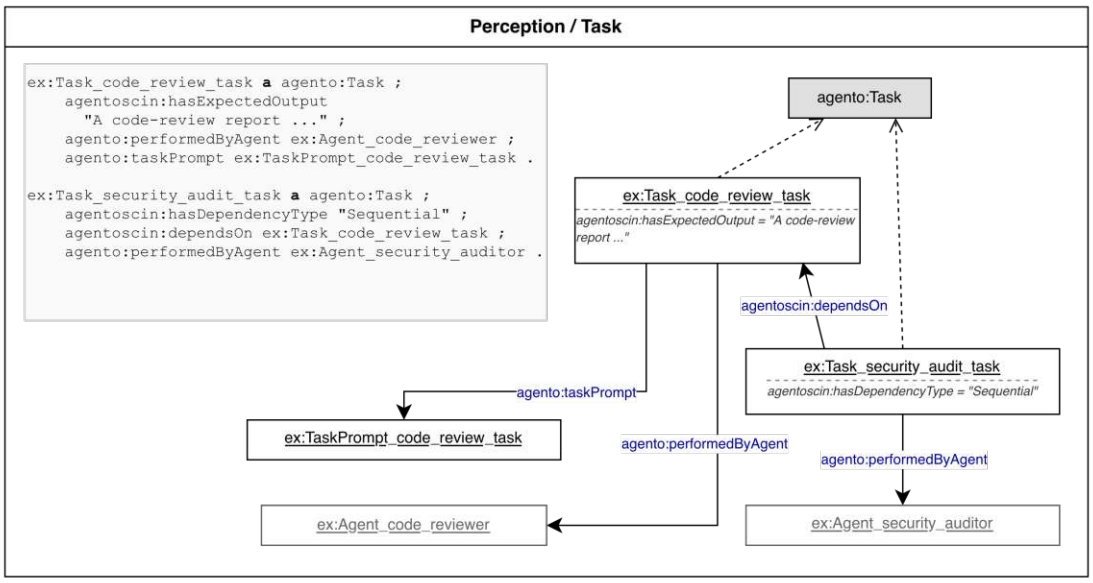

Figure 5.3: Perception and task instantiation excerpt.

The perception and task group captures the units of work an agentic system performs. Its core class `agento:Task` is inherited from AgentO and satisfies R6 directly. AgentOSCIN adds several properties that capture task-level information not represented in AgentO.

The object property `agento:performedByAgent` (R7) records the agent responsible for a task. The associated datatype property `:hasDelegationStrategy` encodes how that assignment was determined: explicit assignment in the source configuration, structural assignment through graph topology, or delegation deferred to runtime. Recording the strategy alongside the assignment preserves a distinction that is otherwise lost when the ontology is queried, since a CrewAI hierarchical process and a CrewAI sequential pro-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

cess produce structurally similar `agento:performedByAgent` assertions for different reasons [Cre25].

The object property `:dependsOn` (R8) links one `agento:Task` to another `agento:Task` it depends on. The datatype property `:hasDependencyType` normalizes the three frameworks' distinct dependency mechanisms into a unified vocabulary. CrewAI's `context` attribute, LangGraph's edges and AutoGen's implicit sequential ordering are heterogeneous on the surface but describe the same underlying relation between tasks [Cre25, Lan25, Mic25]. The `:hasDependencyType` value distinguishes these origins so that the generation direction can produce code in each target framework.

The properties `:hasExpectedOutput` and `:hasOutputSchema` (R9) separate the natural-language description of an expected output from its structured schema. This separation reflects the observation that all three frameworks support the former while only CrewAI consistently supports the latter and that conflating them would force the ontology to either lose information or produce unreliable schemas during generation. The object property `:hasOutputSchema` links a `agento:Task` or an `:Orchestration` to a `:Schema` individual, while `:hasExpectedOutput` is a datatype property carrying the natural-language specification as a string.

`:Schema` is a new class introduced to represent structured-output schemas such as JSON Schema or Pydantic models. Its datatype property `:hasSchemaDefinition` stores the serialised schema content. Modelling the schema as a separate class allows the same schema to be referenced by multiple tasks or by an orchestration and supports future extensions such as schema validation or compatibility checking.

5.3.3 Coordination

The coordination group captures how agents and tasks are organised into a multi-level system, which coordination strategy a unit follows and when its execution terminates. It is the most extensively extended part of the ontology, reflecting the fact that AgentO does not represent coordination patterns or termination semantics as first-class concepts. Five new classes and several supporting properties are introduced.

System Structure The triple `:AgenticSystem`, `:Orchestration` and `agento:Team` (R13) provides the multi-level coordination structure required to represent CrewAI flows containing multiple crews without flattening the hierarchy. `:AgenticSystem` is the top-level container and a subclass of `beam:System`. It is connected to its parts through `:containsAgent`, `:containsTeam` and `:containsOrchestration`. The datatype property `:hasSourceFramework` records which framework the system was extracted from, enabling framework-aware queries and generation.

`:Orchestration` is modeled as a subclass of `agento:Team`, representing the outer flow that orchestrates one or more inner teams via `:orchestratesTeam`. A flat AutoGen team or a single LangGraph graph is represented as an `:AgenticSystem`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

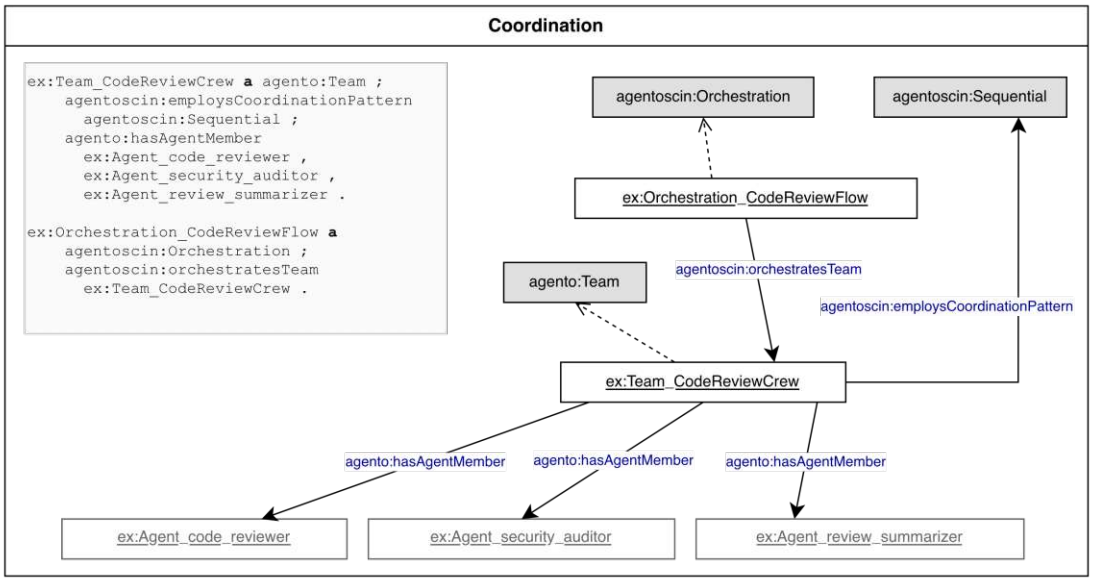

Figure 5.4: Coordination instantiation excerpt.

containing exactly one `agento:Team` and no `:Orchestration`, ensuring that the same ontological structure accommodates both nested and single-level cases.

Coordination Patterns `:CoordinationPattern` (R10) models the coordination strategy as a named concept rather than encoding it as a string-valued property on `agento:Team`. A property-valued encoding would prevent reasoning over the pattern itself, for instance asking whether two teams use compatible coordination strategies or whether a given pattern is supported by a target framework. With `:CoordinationPattern` as a class, individual patterns can be defined as named individuals and queried directly.

The ontology defines eight `:CoordinationPattern` individuals: `:Sequential`, `:Hierarchical`, `:RoundRobin`, `:SelectorBased`, `:Swarm`, `:ReActLoop`, `:Network` and `:Custom`. These cover the coordination strategies observed across all three frameworks. `:Sequential` and `:Hierarchical` map directly to CrewAI's process types. `:RoundRobin`, `:SelectorBased` and `:Swarm` capture AutoGen's team types. `:ReActLoop` represents the tool-calling agent pattern used in LangGraph's `create_react_agent`. `:Network` covers general graph-based coordination and `:Custom` serves as a fallback for patterns not matching any predefined category. The object property `:employsCoordinationPattern` links a `agento:Team` to its pattern.

Workflow Structure `agento:WorkflowPattern` and `agento:WorkflowStep` (R14) are inherited from AgentO and decompose the internal execution structure of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

a coordination unit. AgentOSCIN introduces :ConditionalStep as a subclass of agento:WorkflowStep, attaching the datatype property :hasRoutingLogic at this level. This avoids polluting unconditional steps with a property they cannot use and lets the generator emit unconditional or conditional transitions based on the step's class.

The object property :hasAssociatedAgent links a agento:WorkflowStep to the agento:LLMAgent that executes it, capturing the pattern used in LangGraph where a node function is implemented by a specific agent [Lan25].

Termination Conditions :TerminationCondition (R12) follows the same class-based modelling principle as :CoordinationPattern. AutoGen's termination conditions are first-class objects in the source language and combine using bitwise operators [Mic25]. Representing them as ontology individuals preserves this compositional structure. The object property :hasTerminationCondition links a agento:Team to its termination condition. Five disjoint subclasses cover the observed termination strategies:

- :TurnLimitTermination: execution stops after a maximum number of turns, recorded in :hasMaxTurns (xsd:integer).
- :TaskCompletionTermination: execution stops when all assigned tasks have completed.
- :EventBasedTermination: execution stops when a specific event or condition occurs, described by :hasTriggerExpression (xsd:string).
- :RoutingTermination: execution stops when a routing decision leads to the end of the workflow.
- :CompositeTermination: combines multiple termination conditions via a logical operator. The datatype property :hasOperator records the combinator, for example "AND" or "OR" and the object property :hasSubCondition links to the constituent :TerminationCondition individuals.

5.3.4 Reasoning

The reasoning group captures whether an agent reasons, which pattern its reasoning follows and where the capability originates, as required by R15. AgentOSCIN models this with one new class and several properties on agento:LLMAgent.

:ReasoningPattern is the new class, linked to agents via the object property :employsReasoningPattern. Five subclasses enumerate the patterns observed across the analyzed frameworks (i.e. :ChainOfThought, :ReAct, :ReflectionLoop, :TreeOfThoughts, :Unspecified).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

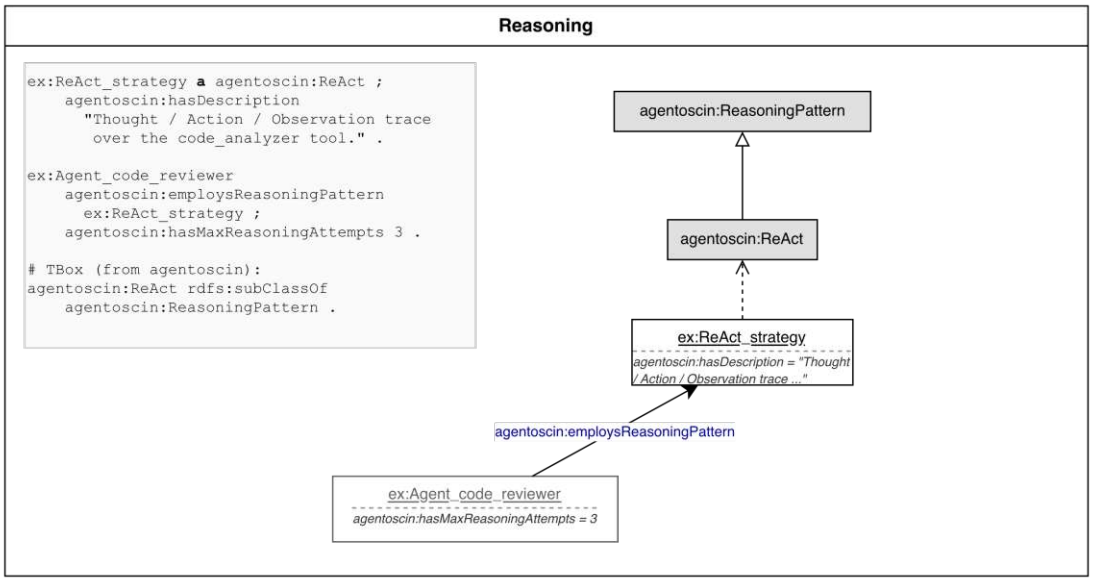

Figure 5.5: Reasoning instantiation excerpt.

The datatype property `:hasReasoningEnabled` (`xsd:boolean`) records whether reasoning is active. `:hasReasoningOrigin` (`xsd:string`) records whether the capability comes from the framework, the underlying language model or the system topology. `:hasMaxReasoningAttempts` (`xsd:integer`) captures an optional upper bound on reasoning iterations. Modelling reasoning origin as a property rather than encoding it implicitly in the agent's class is a deliberate choice driven by the extraction direction. CrewAI exposes reasoning as a configurable agent flag [Cre25], AutoGen leaves it to the underlying model [Mic25] and LangGraph requires it to be constructed in the graph topology [Lan25].

5.3.5 Memory

The memory group captures the scope at which a memory store is attached and the lifetime over which it persists, satisfying R16 (memory scope) and R17 (persistence separation). Its central class `:MemoryBinding` is a relation between a memory owner, which can be an `agento:LLMAgent`, a `agento:Team` or an `:AgenticSystem` and a `agento:Memory` individual. This separation is required because the same memory store can be bound at different scopes in different parts of a system. A CrewAI crew-level memory and a CrewAI agent-level memory can reference the same underlying store but differ in their access scope and a direct property between agent and memory cannot distinguish these cases [Cre25].

The object property `:bindsMemory` connects a `:MemoryBinding` to the `agento:Memory` it wraps. Two specialized sub-properties of the generic `:hasMemoryBinding` refine the attachment point: `:hasAgentMemoryBinding` re-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

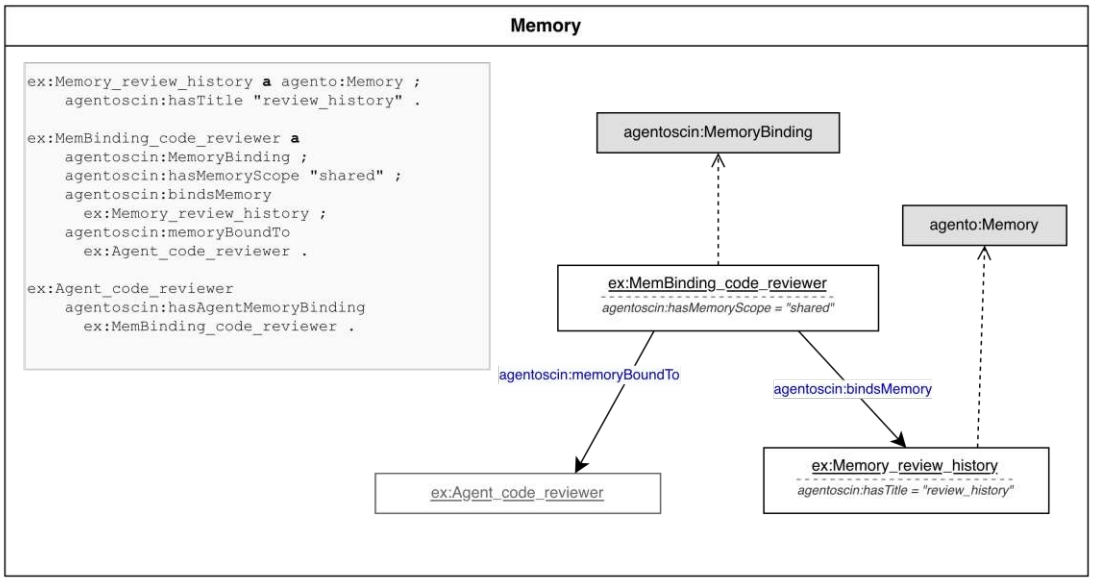

Figure 5.6: Memory instantiation excerpt.

stricts the domain to `agento:LLMAgent` and `:hasTeamMemoryBinding` restricts the domain to `agento:Team`.

The datatype property `:hasMemoryScope` (`xsd:string`) on `:MemoryBinding` records whether the binding is agent-private, group-shared or system-global. The datatype property `:hasPersistenceScope` (`xsd:string`) on `agento:Memory` records whether the memory persists within a single execution, across executions on a thread, or across all sessions. Separating these dimensions preserves the lifecycle distinctions captured in LangGraph's checkpointer-versus-store split while allowing the same dimensional vocabulary to describe CrewAI's and AutoGen's memory configurations.

5.3.6 Tools

The tools group captures the capabilities an agent or task can invoke. The framework-independent tool concept (R18) is largely covered by AgentO's existing `agento:Tool` class. AgentOSCIN adds two datatype properties. `:hasInputSchema` captures the typed input specification that all three frameworks generate from Python type hints, Pydantic models or JSON schemas. `:hasImplementationReference` records a pointer to the tool's implementation, for example a module path or external package identifier, which preserves information needed for code generation when a tool is imported and not defined inline.

The tool assignment scope (R19) is satisfied by distinguishing between two object properties: `agento:agentToolUsage`, inherited from AgentO and `:taskToolUsage`, introduced by AgentOSCIN as a sub-property of `agento:toolUsage`. Modelling the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

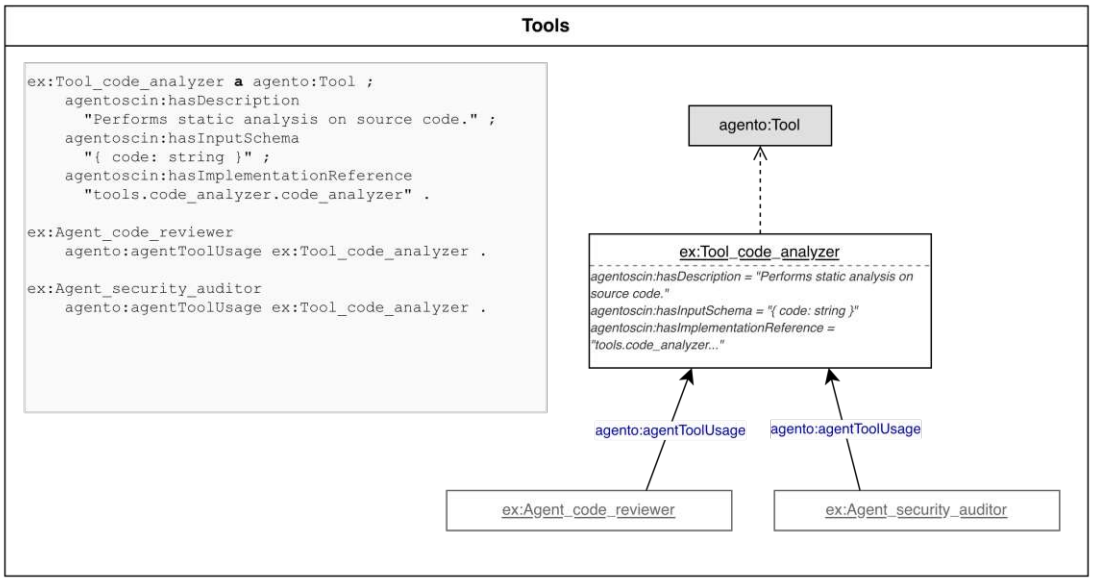

Figure 5.7: Tools instantiation excerpt.

assignment as two separate properties reflects the semantics of the target frameworks: agent-level and task-level tool assignment are not the same relation and conflating them would force the generator to infer scope at code emission time.

5.3.7 Guardrails and Governance

The guardrails and governance group captures validation and human-in-the-loop control over tasks. It introduces two new classes to satisfy R20 and R21, both of which describe concepts entirely absent from AgentO.

`:Guardrail` (R20) attaches to `agento:Task` via the `:hasGuardrail` object property and carries three datatype properties. `:hasGuardrailType` distinguishes function-based, LLM-based and structural validation mechanisms. `:hasValidationLogic` stores the validation expression or function reference. `:hasMaxRetries` (`xsd:integer`) records the maximum number of retry attempts when validation fails. The class-level modelling is necessary because a task may have multiple guardrails chained sequentially [Cre25], which a flat property could not represent.

`:HumanCheckpoint` (R21) attaches to `agento:Task` via `:hasHumanCheckpoint` and carries three datatype properties. `:hasCheckpointType` distinguishes the three observed checkpoint mechanisms: approval, review-and-edit and input-validation. `:hasCheckpointPosition` records where the checkpoint sits in the task execution flow. `:isMandatory` (`xsd:boolean`) indicates whether the checkpoint can be bypassed. Modelling human-in-the-loop as a positional checkpoint rather than as a boolean flag captures the information needed by R21, namely the position of the checkpoint in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

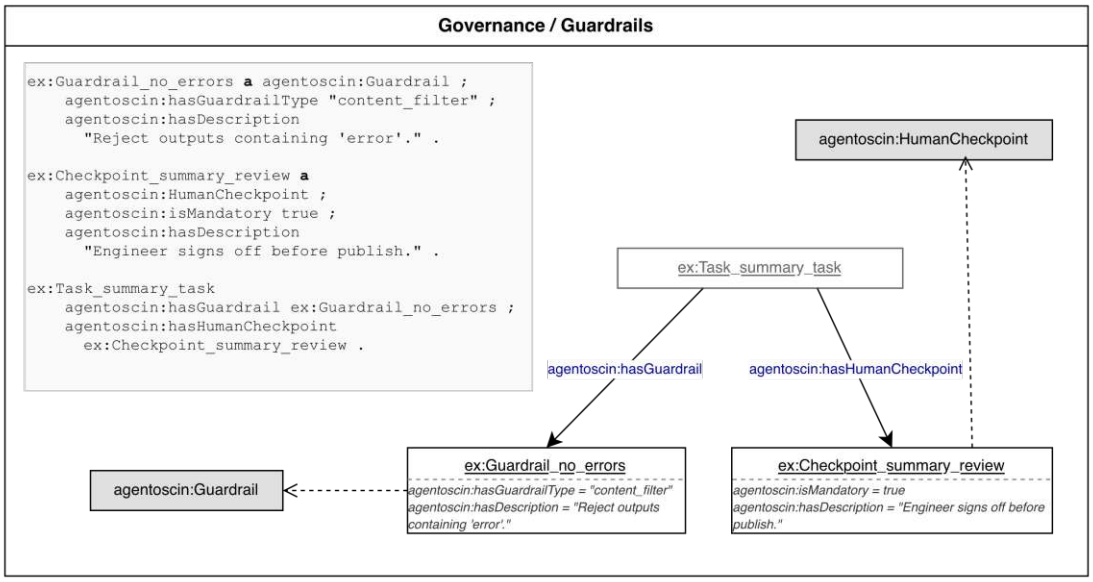

Figure 5.8: Guardrails and governance instantiation excerpt.

the workflow.

The auxiliary class `agento:HumanAgent`, defined in AgentO, extends the agent hierarchy to represent users participating in agentic systems through proxies such as AutoGen's `UserProxyAgent` [Mic25]. The `agento:humanParticipatedIn` relation links a `agento:HumanAgent` to the tasks it interacts with, enabling the ontology to distinguish agent-driven and human-driven contributions to a workflow without reclassifying users as ordinary agents.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

AgenticBIT: Agentic AI Bidirectional Transformation Framework

This chapter presents the transformation framework that links framework-specific source code to the semantic representation developed in Chapter 5. The method builds on the OSCIN framework [dAZS21] for ontology-based source code interoperability, which we extend from an analytical use of the ontology into a bidirectional one. Section 6.2 covers the extraction direction, where three framework-specific extractors turn Python source code into ontology instances. Section 6.3 covers the generation direction, a template-based code synthesiser that turns ontology instances back into executable source code. The two directions are symmetric by design as they share a single intermediate representation and the same ontology.

6.1 Design Principles

Three decisions drive the architecture and account for the concrete design decisions described in the rest of the chapter.

- The transformation is split into three layers, keeping framework knowledge, normalisation logic and ontology knowledge apart. Changes stay local as adding a new framework touches only its parser and generator. Reworking the ontology touches only the population and reading layer.
- Every framework is reduced to one Shared Intermediate Representation (SIR). The SIR is the common language through which the layers communicate and the same SIR dataclasses are used in both directions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- Identifier generating and traversal order are deterministic, so re-running extraction on unchanged input produces an identical output. This is necessary for the diff-based evaluation in Chapter 7, which compares an extracted ABox against a hand-authored ground-truth.

6.2 Extraction: From Source Code to Ontology

In this direction the framework-specific source code patterns are mapped onto a unified semantic representation, which then serves as the basis for the automated transformation into the domain ontology [dAZS21]. The process is organised into three layers (see Fig. 6.1) so that it remains framework-agnostic and extensible.

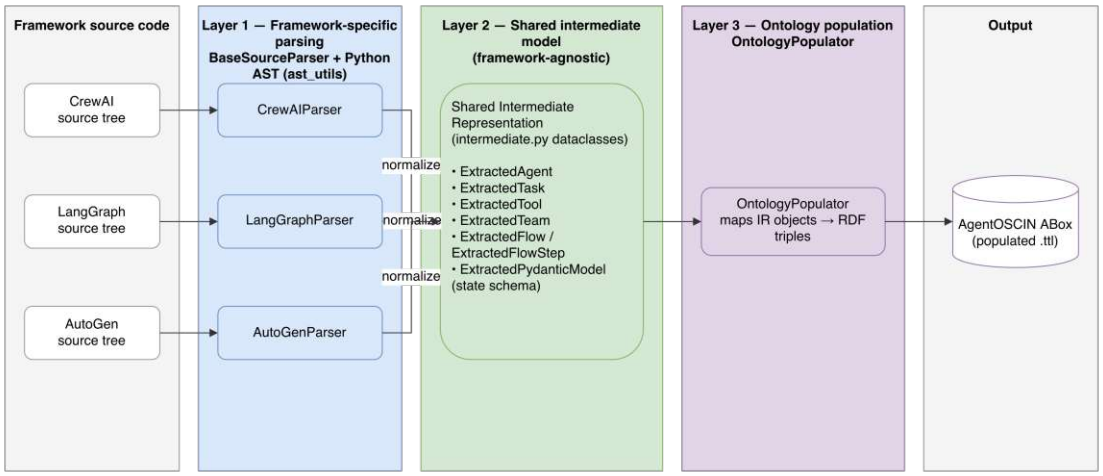

Figure 6.1: OSCIN-based extractor pipeline.

6.2.1 Framework-Specific Parsing via Static Analysis (Layer 1)

The first layer uses static program analysis to interpret the syntactic structure of the source code. Each parser uses Python's built-in `ast` module to construct an Abstract Syntax Tree of the target files and traverses it with the Visitor pattern (via `ast.walk` or `ast.NodeVisitor`) to find framework-specific structures [Pyt24].

The main difficulty this layer has to resolve is that the three frameworks express the same agentic constructs through very different syntactic implementations. CrewAI is declarative and relies on decorators (`@CrewBase`, `@agent`, `@task`, `@crew`) backed by external YAML configuration [Cre25]. AutoGen is imperative, where agents and teams are plain constructor calls such as `AssistantAgent(...)` and `RoundRobinGroupChat(...)` [Mic25]. LangGraph is builder-based and expresses orchestration as method calls on a `StateGraph` object (`add_node`, `add_edge`, `add_conditional_edges`) [Lan25]. The parsers work as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- The `CrewAIParser` resolves decorator names against the AST, detects `@agent` and `@task` methods and binds the YAML-declared `role`, `goal` and `backstory` to the corresponding method.
- The `AutoGenParser` scans for variable assignments whose value is an `AssistantAgent` instantiation and extracts the constructor arguments.
- The `LangGraphParser` performs a two-pass walk. The first pass locates every `StateGraph` instantiation and its state schema and the second pass routes each subsequent builder call into the state of the graph variable it was invoked on. This two-pass design preserves the back-edges and loops that a single-pass scan would lose.

6.2.2 Shared Intermediate Representation (Layer 2)

To keep the extraction logic separate from the ontology-population logic, Layer 1 maps every framework-specific abstraction onto the SIR. The SIR is a set of strictly typed Python dataclasses, namely `ExtractedAgent`, `ExtractedTask`, `ExtractedTool`, `ExtractedTeam`, `ExtractedFlow`, `ExtractedFlowStep` and `ExtractedPydanticModel`, that act as an ontological proxy and normalise all frameworks into one schema. Both an AutoGen `AssistantAgent` and a LangGraph `add_node()` invocation are reduced to a single `ExtractedAgent` that captures only the semantic essence the ontology needs, such as the role, goal, tools and language model.

The field names align with ontology concepts and not with the terminology of any framework and each field is annotated in-source with the property it maps to (see Fig. 6.2). For example `ExtractedAgent.backstory` maps to `agentPrompt.promptContext` and `ExtractedAgent.llm` maps to `useLanguageModel`. Where a framework uses different wording, such as AutoGen's `system_message` instead of `backstory`, the parser performs the renaming. So the mismatch gets resolved right at the parser boundary.

```python
@dataclass
class ExtractedAgent:
    """
    Intermediate representation of an agent from any framework.

    Mapping table: Section 1 (Agent Mapping).
    """

    # Identity
    agent_key: str  # Unique key within the extraction run
    role: str  # → agentID, agentRole
    goal: str  # → hasAgentGoal
    backstory: str  # → agentPrompt.promptContext

    # Optional parameters
    llm: Optional[str] = None  # → useLanguageModel
    tools: list[str] = field(default_factory=list)  # → agentToolUsage
    reasoning: bool = False  # → hasReasoningEnabled
    max_reasoning_attempts: Optional[int] = None  # → hasMaxReasoningAttempts
    memory: bool = False  # → hasMemoryBinding
    verbose: Optional[bool] = None  # → hasAgentConfig
    allow_delegation: Optional[bool] = None  # → hasAgentConfig
    knowledge_sources: list[str] = field(default_factory=list)  # → hasKnowledge

```

Figure 6.2: Excerpt of the `ExtractedAgent` dataclass. Each field is named after the ontology concept it denotes and annotated in-source with the property it maps to.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

This is what gives the architecture its extensibility. Adding a fourth framework requires only a new parser that emits SIR objects and a change to the ontology touches only the SIR-to-TBox mapping and never the individual parsers.

6.2.3 Ontology Population and Semantic Assertion (Layer 3)

The final layer iterates over the SIR objects to populate the AgentOSCIN ontology using `rdflib` [RDF25]. It reads only from the framework-neutral intermediate objects and never touches the AST or the YAML. This keeps all semantic-assertion logic in one framework-independent module. The populator produces a deterministic and stable URI for each construct, derived from its sanitised key, and asserts the RDF triples that enforce the schema defined in the TBox (see Listing 6.1).

Beyond flat type assertions the populator reifies the relationships the frameworks leave implicit. When it instantiates an agent it creates an individual of type `:LLMAgent`, promotes the backstory to a first-class `:Prompt` individual linked through `:agentPrompt` and asserts `:useLanguageModel` to the designated LLM. In the same way a coordination style such as `sequential` or `hierarchical` is asserted as an explicit coordination-pattern individual. Because the identifiers are found deterministically the resulting ABox is stable across runs and can be compared reliably against a hand-authored ground-truth.

```
for agent in sir.agents:
    agent_uri = EX[f"Agent_{sanitize(agent.name)}"]

    # Assert class type and role
    graph.add((agent_uri, RDF.type, agento.LLMAgent))
    graph.add((agent_uri, agento.agentRole, Literal(agent.role)))

    # Reify the backstory as a linked Prompt individual
    if agent.backstory:
        prompt_uri = EX[f"AgentPrompt_{sanitize(agent.name)}"]
        graph.add((prompt_uri, RDF.type, agento.Prompt))
        graph.add((prompt_uri, agento.promptContext, Literal(agent.
            backstory)))
        graph.add((agent_uri, agento.agentPrompt, prompt_uri))
```

Listing 6.1: Ontology population: mapping SIR objects to OWL individuals with RDFLib, including reification of the backstory as a `:Prompt`.

Running an extraction over CLI The extraction stage is invoked through the CLI by pointing it at a directory of framework-specific source files. Listing 6.2 shows an example that parses a CrewAI implementation of the maths system into an `AgentOSCIN` ABox serialised as Turtle.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
# Extract CrewAI source files -> Turtle ABox
python -m oscin.cli extract examples/parallel/maths/crewai/
    source_files \
     --framework crewai \
     --system-name MathsSystem \
     --namespace "http://example.org/maths#" \
     --output output/maths/crewai_ontology.ttl
```

Listing 6.2: Extraction: parsing CrewAI source code into an `AgentOSCIN` ontology instance (`.ttl`).

6.3 Generation: From Ontology to Source Code

For the generation direction the scope of OSCIN's final phase was extended. Rather than querying the ontology for analytical purposes as in the original OSCIN method of de Aguiar et al. [dAZS21], we use the semantic graph as an intermediary for bidirectional interoperability and translate the populated AgentOSCIN ontology back into executable source code for the agentic AI frameworks. The generation pipeline (see Fig. 6.3) is the inverse of extraction and runs through the same three layers in reverse.

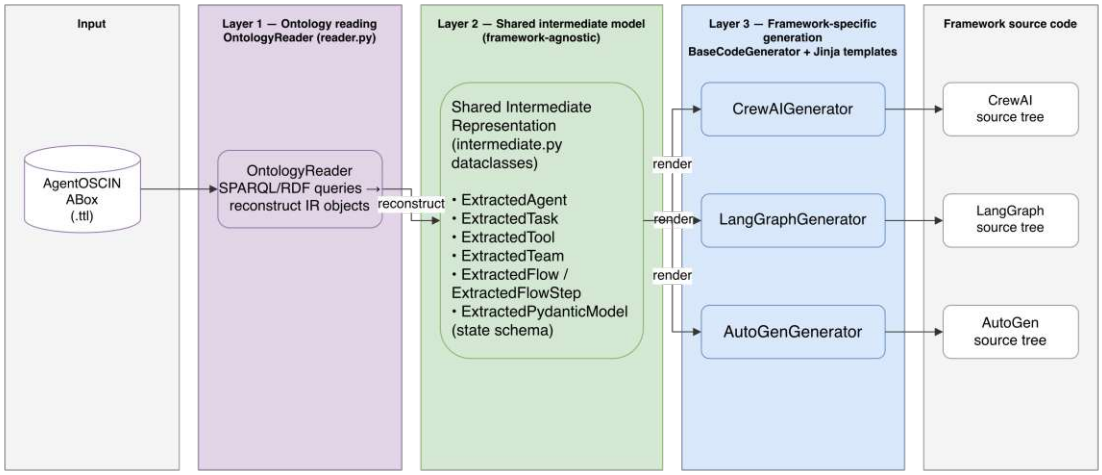

Figure 6.3: OSCIN-based code-generator pipeline.

6.3.1 Ontology Reading and IR Reconstruction (Layers 1 & 2)

A framework-agnostic reader uses `rdflib` to parse the serialised ontology, for example a Turtle file, walks the ABox to find instances of the core conceptual classes (`:LLMAgent`, `:WorkflowStep` and so on) and resolves their object properties. The reader is the structural inverse of the populator and reconstructs the same SIR dataclasses a parser would have produced. When it reads an `:LLMAgent`, it follows `:agentPrompt` to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

recover the backstory and `:useLanguageModel` to recover the LLM configuration. Since the intermediate contract is shared verbatim across both directions the only asymmetry lives in the leaf layers. Extraction has to recover structure from heterogeneous syntax, whereas generation has to re-impose it.

6.3.2 Framework-Specific Code Generation (Layer 3)

Dedicated generators (`CrewAIGenerator`, `AutoGenGenerator` and `LangGraphGenerator`) consume the populated SIR and render it into source code. Every generator owns a Jinja2 template (`crewai_main.py.j2`, `autogen_main.py.j2` and `langgraph_main.py.j2`). An example template excerpt can be seen in Figure 6.4.

For frameworks with split configuration the generator also produces the supporting artefacts the runtime expects. CrewAI receives an execution script along with `agents.yaml` and `tasks.yaml` and LangGraph receives a separate tools module alongside its graph script. Because both directions attach to the same SIR and ontology a system extracted from one framework can be regenerated into any other, which is the bidirectional-interoperability property this transformation framework aims to achieve.

```
# -- Agents --
{% for agent in agents %}
{{ agent.var_name }} = AssistantAgent(
    name="{{ agent.name }}",
    model_client=model_client,
    {% if agent.tool_vars %}
    tools=[{{ agent.tool_vars | join(', ') }}],
    {% endif %}
    {% if agent.memory_class %}
    memory=[{{ agent.memory_class }}()],
    {% endif %}
    {% if agent.description %}
    description="{{ agent.description }}",
    {% endif %}
    system_message=(
        "{{ agent.system_message }}"
    ),
)
```

Figure 6.4: Excerpt of the AutoGen Jinja2 template, emitting one `AssistantAgent` per SIR agent.

Running a generation over CLI The generation stage takes an ontology instance and synthesises runnable source code for a chosen target framework. Listing 6.3 shows both the same framework case and the cross-framework case in which a CrewAI derived ABox is used to generate AutoGen code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
# Same-framework: ontology -> CrewAI code
python -m oscin.cli generate output/maths/crewai_ontology.ttl \
    --target-framework crewai \
    --output-dir generated/maths/crewai/

# Cross-framework: CrewAI-derived ontology -> AutoGen code
python -m oscin.cli generate output/maths/crewai_ontology.ttl \
    --target-framework autogen \
    --output-dir generated/maths/autogen/
```

Listing 6.3: Generation: synthesising source code from an ontology instance, either same-framework or cross-framework (CrewAI ABox → AutoGen code).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Evaluation

This chapter presents the evaluation results for both AgentOSCIN and AgenticBIT. The evaluation of AgentOSCIN begins with a structural analysis using the Ontology Pitfall Scanner [PVGPSF14], followed by a competency coverage assessment [GF95] via requirements-driven CQs to compare its performance against the AgentO and DBM baselines. Next, the operational usability of the artefact is evaluated using our OSCIN-based AgenticBIT framework.

7.1 Ontology Evaluation

AgentOSCIN was evaluated in two ways. First, the structure of the ontology was assessed using the OOPS! ontology evaluation tool [PVGPSF14], which detects common modelling pitfalls and categorizes them as critical, important or minor. Second, the ontology was verified against the competency questions collected during ontology engineering (see Table 7.9), checking whether a populated ontology provides sufficient vocabulary to satisfy the functional and non-functional requirements identified for the artefact. The competency questions are derived directly from the requirement specifications established in Chapter 4.

Metric	Count
Axiom	679
Logical axiom count	254
Declaration axioms count	168
Class count	52
Object property count	60
Data property count	42
Individual count	8
Annotation property count	9

Table 7.1: AgentOSCIN ontology structural metrics, as reported by Protégé.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

7.1.1 Structural Ontology Evaluation via OOPS!

The ontology was scanned with the OOPS! Pitfall Scanner, once in its initial state and once after the corrections described below. Figure 7.1 reports the initial scan and Figure 7.2 the scan after the corrections were applied. No critical pitfalls were detected in either run.

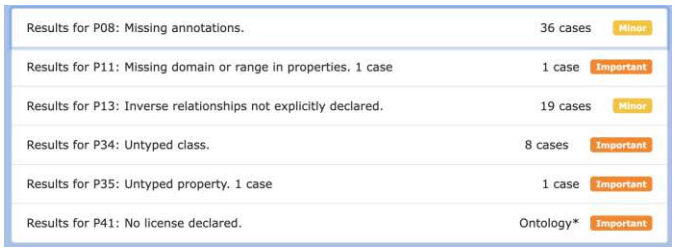

Figure 7.1: Results of the OOPS! Pitfall Scanner before the corrections

Figure 7.2: Results of the OOPS! Pitfall Scanner after the corrections

The initial scan reported four important and two minor pitfalls, several of which originate from the vocabularies that AgentOSCIN reuses. The eight cases of "P34: Untyped class" and the single case of "P35: Untyped property" correspond to entities referenced from the vocabularies imported through agentO, which are typed in their source ontologies and not redeclared in AgentOSCIN. The scanner evaluates the submitted document in isolation and does not resolve `owl:imports`, so entities that are referenced locally but declared in an imported ontology are reported as untyped. Both pitfalls are therefore artefacts of the evaluation setup rather than defects of the ontology.

Two further pitfalls were addressed directly. The single case of "P11: Missing domain or range in properties" concerned the object property `hasMemoryBinding`, which declared a range but no domain. A domain was added as the union of `AgenticSystem`, `LLMAgent` and `Team`, mirroring the range already declared for `memoryBoundTo` and consistent with the domains of its two subproperties. "P41: No licence declared" was resolved by attaching a CC BY 4.0 licence to the ontology using `dcterms:licence`. Neither pitfall appears in the second scan.

Of the thirty-six cases of "P08: Missing annotations", thirty-five concerned terms minted by AgentOSCIN, predominantly the datatype properties introduced for framework mapping together with the reasoning pattern and coordination pattern taxonomies, which carried an `rdfs:label` but no `rdfs:comment`. A textual definition was supplied for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

each of these terms. The single remaining case concerns `beam:System`, which is reused through agentO and documented in its source ontology, and is thus of the same kind as the untyped entity cases discussed above.

The minor pitfall "P13: Inverse relationships not explicitly declared", reported in nineteen cases, was not addressed. The transformation pipeline traverses these relations in a single direction only, so declaring the inverses would add no information that is queryable through the competency questions.

After the corrections the scanner reports one case of P08, nineteen of P13, eight of P34 and one of P35, and no critical pitfalls (Figure 7.2). Apart from the P13 cases, which reflect a deliberate modelling decision, every remaining case concerns an entity that is declared or documented in an ontology imported through agentO rather than in AgentOSCIN itself.

In order to address the import-related findings and to support future iteration on and contribution to the vocabulary, a standalone version of AgentOSCIN was produced and is available in the accompanying repository[1]. This version inlines the axioms of agentO rather than referencing them through `owl:imports`, so that every entity used in the ontology is declared within the document itself. Terms originating in agentO retain their original IRIs, which preserves the provenance of the reused axioms and keeps the extension relationship explicit. The standalone version adopts the namespace `https://w3id.org/agentoscin`, which replaces the editor-generated namespace used during development and provides a persistent identifier suitable for publication and reuse.

Because the namespace differs from the one used throughout the evaluation, the standalone version was not itself subject to the evaluation reported in this chapter. Every ground-truth graph, competency question and transformation template in the evaluation corpus refers to terms in the development namespace, so substituting the standalone version would require a corresponding renaming across the entire corpus and would invalidate the comparability of the reported measurements. The standalone version is therefore offered as the artefact intended for onward publication and reuse, while the version evaluated here remains the reference for the results presented in Chapter 7.

7.1.2 Ontology Evaluation via Competency Questions

To provide a robust testing environment a single knowledge graph that loads both ontologies (i.e. AgentO and AgentOSCIN) together with the hand-authored ABoxes of the six original systems of the evaluation corpus (see Tab. 7.11) was assembled. Each was taken from the official website or GitHub repository of its framework (joke and maths from LangGraph, code-review and travel-planning from AutoGen, tech-blog and meeting-assistant from CrewAI) and was changed substantially over the course of this

[1] `https://github.com/danilipp17/thesis_code/blob/main/ontology/agentoscin-standalone.ttl`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

research (e.g. the notebooks were converted from `.ipynb` to `.py` modules so they could be consumed by the pipeline, framework and library calls were updated to the versions used in this work, and some examples that were removed from the upstream repositories during the research were retained and adapted here). The adapted versions used here are available in the accompanying repository[2]. Because some of the ontology features are not exercised by any of these six systems (reasoning patterns, memory bindings and scopes, validation guardrails and human checkpoints), one small synthetic feature-demo system[3] whose purpose is to instantiate those remaining concepts was populated, so that every competency question can be evaluated against at least one concrete instance.

For each of the seven concept groups, one representative SPARQL query and the answer it returns over this knowledge graph are shown: Agent Definition (see 7.1), Perception/Task (see 7.2), Coordination (see 7.3), Reasoning (see 7.4), Memory (see 7.5), Tools (see 7.6) and Guardrails/Governance (see 7.7). The complete set of competency-question queries is available online[4].

[2] https://github.com/danilipp17/thesis_code/tree/main/examples

[3] https://github.com/danilipp17/thesis_code/blob/main/ontology/feature_demo.ttl

[4] https://github.com/danilipp17/thesis_code/blob/main/ontology/competency_queries.rq

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Agent Def. CQ-2 (R1) — What identifier and role is assigned to each agent?

```
SELECT ?agent ?id ?role WHERE {
    ?agent a agento:LLMAgent .
    OPTIONAL { ?agent agento:agentID   ?id   }
    OPTIONAL { ?agent agento:agentRole ?role }
}
ORDER BY ?agent
```

Listing 7.1: SPARQL query for CQ-2

Agent	agentID	agentRole
Agent_researcher	researcher	Senior Tech Researcher
Agent_writer	writer	Tech Blog Writer
Agent_editor	editor	Content Editor
Agent_Code_Reviewer	Code_Reviewer	Code_Reviewer
Agent_meeting_analyzer	meeting_analyzer	Meeting Transcript Analysis Agent
Agent_generate_joke	generate_joke	generate_joke

Table 7.2: Answer to CQ-2 (excerpt; 18 agents in total across the seven systems).

Perception/Task Def. CQ-13 (R7) — How was a task's assignment determined (delegation strategy)?

```
SELECT ?task ?delegationStrategy WHERE {
    ?task a agento:Task ;
          agentoscin:hasDelegationStrategy ?delegationStrategy .
}
ORDER BY ?task
```

Listing 7.2: SPARQL query for CQ-13

Task	delegationStrategy
Task_research_topic	ExplicitAssignment
Task_analyze_meeting	ExplicitAssignment
Task_task_generate_joke	TopologyDetermined
Task_autogen_task_0	OrchestratorDelegated

Table 7.3: Answer to CQ-13 (excerpt). The strategy distinguishes explicitly assigned CrewAI tasks, topology-determined LangGraph nodes and orchestrator-delegated AutoGen team calls.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Coordination CQ-23 (R14) — What is the step-by-step execution structure of a team's workflow?

```
SELECT ?pattern ?step ?order ?nextStep WHERE {
    ?pattern a agento:WorkflowPattern ;
             agento:hasWorkflowStep ?step .
    OPTIONAL { ?step agento:stepOrder ?order    }
    OPTIONAL { ?step agento:nextStep  ?nextStep }
}
ORDER BY ?pattern ?order
```

Listing 7.3: SPARQL query for CQ-23

WorkflowPattern	Step	order	nextStep
StateGraph (joke)	generate_joke	1	— (conditional)
StateGraph (joke)	improve_joke	2	polish_joke
StateGraph (joke)	polish_joke	3	—
MeetingFlow	load_meeting_notes	1	generate_tasks
MeetingFlow	generate_tasks	2	{trello, csv, slack}

Table 7.4: Answer to CQ-23 (excerpt). The branch from `generate_joke` is conditional (no fixed `nextStep`); the meeting-assistant flow fans out to three parallel sinks at order 2.

Reasoning CQ-26 (R15) — What reasoning pattern does an agent follow?

```
SELECT ?agent ?pattern ?patternType WHERE {
    ?agent agentoscin:employsReasoningPattern ?pattern .
    ?pattern a ?patternType .
    FILTER(?patternType != agentoscin:ReasoningPattern)
    FILTER(?patternType != owl:NamedIndividual)
}
ORDER BY ?agent
```

Listing 7.4: SPARQL query for CQ-26

Agent	pattern	patternType
Agent_planner	ReAct_planner	agentoscin:ReAct

Table 7.5: Answer to CQ-26. Reasoning patterns are not surfaced by the six corpus systems and are exercised here through the feature-demo system.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Memory CQ-28 (R16) — Is an agent's memory private, shared with the team or accessible system-wide?

```
SELECT ?owner ?ownerKind ?memory ?scope WHERE {
    ?owner agentoscin:hasMemoryBinding ?binding .
    ?binding agentoscin:bindsMemory ?memory .
    OPTIONAL { ?binding agentoscin:hasMemoryScope ?scope }
    BIND(
        IF(EXISTS { ?owner a agento:LLMAgent },          "agent-
           private",
        IF(EXISTS { ?owner a agento:Team },              "team-
           shared",
        IF(EXISTS { ?owner a agentoscin:AgenticSystem }, "system-
           wide",
                                                         "other")))
        AS ?ownerKind)
}
```

Listing 7.5: SPARQL query for CQ-28

owner	ownerKind	memory	scope
Agent_planner	agent-private	PlannerScratchpad	private
feature_demo	system-wide	GlobalMemory	global

Table 7.6: Answer to CQ-28. The `ownerKind` is derived from the class of the binding's owner, distinguishing agent-private from system-wide memory.

Tools CQ-31 (R18) — What input does each tool expect?

```
SELECT ?tool ?inputSchema WHERE {
    ?tool a agento:Tool ;
          agentoscin:hasInputSchema ?inputSchema .
}
ORDER BY ?tool
```

Listing 7.6: SPARQL query for CQ-31

Tool	inputSchema (abbreviated)
Tool_add	{a: integer, b: integer}
Tool_subtract	{a: integer, b: integer}
Tool_code_analyzer	{code: string, language: string}
Tool_search	{query: string}

Table 7.7: Answer to CQ-31 (excerpt; 6 tools in total). Input schemas are stored as serialised JSON schema literals.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Guardrails/Governance CQ-34 (R20) — Does a task have validation guardrails and what type of check do they perform?

```
SELECT ?task ?guardrail ?guardrailType ?validationLogic ?maxRetries
    WHERE {
    ?task agentoscin:hasGuardrail ?guardrail .
    OPTIONAL { ?guardrail agentoscin:hasGuardrailType   ?
       guardrailType   }
    OPTIONAL { ?guardrail agentoscin:hasValidationLogic ?
       validationLogic }
    OPTIONAL { ?guardrail agentoscin:hasMaxRetries      ?maxRetries
              }
}
```

Listing 7.7: SPARQL query for CQ-34

task	guardrailType	validationLogic	maxRetries
Task_plan	schema_validation	Output must contain a numbered step.	3

Table 7.8: Answer to CQ-34.

7.1.3 CQ Coverage Results

Each competency question was classified into one of three coverage levels for a given ontology. Full (✓) means the ontology expresses every concept and property the question requires, so the query returns complete answers. Partial (∼) means the ontology can represent the subject entity of the question and the question could in principle be answered, but at least one of the qualifying details it asks for is not present in the knowledge graph under that vocabulary (e.g. the entity can be located but the discriminating property (a memory scope, a tool input schema, an agent type) lives only in the extension and therefore has no binding.) None (×) means the central concept of the question is not expressible at all.

Running the SPARQL queries over AgentOSCIN, every one of the 35 competency questions resolves to full coverage. The extension captures the structural and behavioural depth required by all three frameworks, giving a Competency Question Coverage of 100 %.

As a comparative baseline we re-ran the same questions restricted to the AgentO vocabulary only and additionally assessed a unified class model [DBM25]. The DBM model is a UML class diagram, so its coverage is judged under the assumption that the diagram were instantiated faithfully as an ABox and queried in the same way. A question is counted as covered if the diagram's classes and relations could supply the answer. Table 7.10 reports the headline numbers and Table 7.9 the per-CQ breakdown.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 7.9: Per-CQ coverage. ✓ full, ∼ partial, × none.

CQ	Req.	Concept	Competency Question	AgentO	DBM	AgentOSCIN
CQ-1	R1	Agent Def.	Which agents are defined in this system?	∼	∼	✓
CQ-2	R1	Agent Def.	What identifier and role is assigned to each agent?	✓	×	✓
CQ-3	R1	Agent Def.	What category (worker / manager / human-proxy) does the agent declare?	×	×	✓
CQ-4	R2	Agent Def.	Does the agent distinguish orchestration-layer from language-model prompts?	✓	×	✓
CQ-5	R2	Agent Def.	What explicit directive role does each prompt declare?	×	×	✓
CQ-6	R3	Agent Def.	What are the instruction, context and output indicator components of a given agent's prompt?	✓	×	✓
CQ-7	R4	Agent Def.	Which core agent properties (id, role, language model, prompt, goal) are populated?	✓	∼	✓
CQ-8	R4	Agent Def.	Which extension properties (type, reasoning pattern, memory binding) are populated?	×	∼	✓
CQ-9	R5	Agent Def.	Is a given agent a proxy for human input?	∼	×	✓
CQ-10	R5	Agent Def.	Is a given agent classified as worker or manager?	×	×	✓
CQ-11	R6	Perception	What tasks does this system define and what does each task instruct the assigned agent to do?	∼	∼	✓
CQ-12	R7	Perception	Which agent is responsible for a given task?	✓	✓	✓
CQ-13	R7	Perception	How was that assignment determined (delegation strategy)?	×	×	✓
CQ-14	R8	Perception	Which tasks must complete before a given task can start?	×	✓	✓
CQ-15	R8	Perception	What kind of dependency holds between a task and its predecessor (ordering only, or output-passing)?	×	∼	✓
CQ-16	R9	Perception	What output is a task expected to produce?	×	×	✓
CQ-17	R9	Perception	Is a structured output schema defined for the task?	×	×	✓
CQ-18	R10	Coordination	What coordination pattern does a given team follow?	×	×	✓
CQ-19	R11	Coordination	Which agents belong to a given team?	✓	×	✓
CQ-20	R11	Coordination	How are those agents bound to the team's coordination structure?	∼	×	✓
CQ-21	R12	Coordination	Under what conditions does a team's execution stop?	×	×	✓
CQ-22	R13	Coordination	Does a team contain subteams and if so how are they nested?	×	×	✓
CQ-23	R14	Coordination	What is the step-by-step execution structure of a team's workflow?	✓	×	✓
CQ-24	R14	Coordination	Where does the workflow branch conditionally?	×	×	✓
CQ-25	R15	Reasoning	Does the agent use reasoning at all?	×	∼	✓

Continued on next page

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CQ	Req.	Concept	Competency Question	AgentO	DBM	AgentOSCIN
CQ-26	R15	Reasoning	What reasoning pattern does it follow?	×	∼	✓
CQ-27	R15	Reasoning	Where does that reasoning capability originate?	×	×	✓
CQ-28	R16	Memory	Is an agent's memory private, shared with the team or accessible system-wide?	∼	∼	✓
CQ-29	R17	Memory	Does a given memory persist only during execution, across threads or across all sessions?	∼	∼	✓
CQ-30	R18	Tools	What tools are available in this system?	∼	✓	✓
CQ-31	R18	Tools	What input does each tool expect?	∼	×	✓
CQ-32	R19	Tools	Is a given tool bound at the agent level?	✓	✓	✓
CQ-33	R19	Tools	Is a tool bound or overridden for a specific task?	∼	✓	✓
CQ-34	R20	Guardrails	Does a task have validation guardrails and what type of check do they perform?	×	✓	✓
CQ-35	R21	Guardrails	At which points in the workflow must a human provide input or approval?	×	×	✓

Ontology / model	Full	Partial	None	Full %	Full+Partial %
AgentOSCIN	35	0	0	100.0	100.0
AgentO	8	9	18	22.9	48.6
DBM	6	9	20	17.1	42.9

Table 7.10: Competency Coverage across all data models

The ontology evaluation shows that neither AgentO nor the DBM unified model provides sufficient vocabulary to semantically represent agentic AI systems at the level of detail captured by our requirements elicitation. AgentO answers 8 of 35 questions fully and a further 9 only partially, leaving 18 entirely unanswerable. The DBM model answers 6 fully and 9 partially, with 20 unanswerable. The two baselines have different gap profiles. The DBM model adds task dependencies, task-level tool binding and guardrail validation that AgentO lacks (CQ-14, CQ-33, CQ-34), while losing AgentO's representation of agent identity, prompt components and team membership (CQ-2, CQ-6, CQ-19). Both leave the same agentic-system concepts uncovered: coordination patterns, termination conditions, conditional workflow branches, reasoning patterns and human-in-the-loop checkpoints. AgentOSCIN raises full coverage from 22.9 % (AgentO) and 17.1 % (DBM) to 100 %, which demonstrates that the extension delivers the missing vocabulary needed to bridge heterogeneous frameworks without losing information.

7.1.4 Summary of Ontology Evaluation

The achievement of 100 % full concept coverage demonstrates that the ontology possesses a sufficiently expressive vocabulary to capture the structural and behavioural depth of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the three analysed frameworks. Comparative analysis with both AgentO and the DBM unified model indicates that without the extensions introduced in our work, relevant agentic features cannot be modelled at a depth sufficient to capture the underlying source code logic.

The structural evaluation with OOPS! supports this result from a modelling-quality perspective. No critical pitfalls were detected and the remaining important and minor pitfalls either originate from the imported vocabularies or were corrected directly, leaving the AgentOSCIN extensions free of structural defects.

While these results establish AgentOSCIN as a robust theoretical framework for the framework-agnostic representation of heterogeneous agentic AI implementations, this evaluation does not yet account for its operational usability. To address this, the following section describes a series of controlled transformation experiments conducted to evaluate whether AgentOSCIN can effectively serve as a semantic intermediate layer to enable interoperability between modern agentic AI frameworks.

7.2 Transformation Experiments

This section reports the five transformation experiments that exercise the AgenticBIT pipeline end to end on the evaluation corpus (see Tab. 7.11). The first three experiments hold the framework constant and measure how faithfully the extractor recovers an ABox from source code (as well as a comparison to an LLM-based baseline) and how faithfully the generator emits source code from an ABox. The fourth experiment crosses a framework boundary and measures translation through the AgenticBIT Framework. A fifth experiment evaluates the runtime behaviour of the generated code.

Experiment Corpus Each family is anchored by an original implementation that was taken and adapted from the framework's website and repository (CrewAI, LangGraph and AutoGen respectively) and then manually transformed to the remaining two frameworks; the implementations used here are available in the accompanying repository[5]. The ports preserve agent roles, prompts, tool usage and the intended coordination pattern, but adopt the conventions of the target framework. CrewAI variants use the Flow, CrewBase and YAML configuration pattern. AutoGen variants use `RoundRobinGroupChat` or `SelectorGroupChat` with the appropriate termination condition. LangGraph variants use a typed `StateGraph`. To confirm that the ports preserve the goals of the original system each family was executed and its runtime behaviour compared. All ports produced the expected behaviour. Each cell is therefore paired with two semantic siblings, which lets the same task be observed in three frameworks under identical evaluation conditions.

[5] https://github.com/danilipp17/thesis_code/tree/main/examples/parallel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Family	Origin	Description
joke	LangGraph	Generate, improve and polish a joke with a conditional gate that retries weak drafts.
code-review	AutoGen	Reviewer, security auditor and summariser inspect a code snippet and produce a structured report.
tech-blog	CrewAI	Researcher, writer and editor collaborate to draft and refine a short technical blog post.
meeting-assistant	CrewAI	Load a meeting transcript, extract actionable tasks, then publish them to Trello, CSV and Slack.
travel-planning	AutoGen	Selector-coordinated team of a planner, a local guide and a summariser produces a travel itinerary.
maths	LangGraph	Single agent with arithmetic tools answers maths tasks

Table 7.11: Original source code examples.

Ground-Truth Ontology Instance Each of the eighteen cells ships with a hand-authored ABox in Turtle syntax. The ABox encodes the intended semantics of the source code following the conventions established for AgentOSCIN and captures the framework-specific patterns that distinguish each implementation. These ground-truths serve three roles. First, the reference against which extracted ABoxes are evaluated. Second, the input artefact for the generation experiments and third, the target for the cross-framework experiment.

A separate ground-truth was authored for every cell rather than one shared ABox per family. This follows from the design of AgentOSCIN. To support bidirectional transformation the ontology must be expressive enough to represent the framework-specific abstractions of each target, such as the CrewAI Flow, the AutoGen group chat and termination conditions and the typed LangGraph StateGraph. These abstractions are modelled deliberately and are not fully interchangeable across frameworks. This means that although the three ports of a family pursue the same goals their faithful representations will differ. Authoring one ground-truth per cell measures every implementation against its own intended semantics. The cross-framework experiments test how well the semantic representation is translated across frameworks.

7.2.1 Extraction Experiments

Experiment Setup. The extraction experiment runs the extractor on each cell's source tree and compares the resulting ABox against the hand-authored ground-truth ABox that ships with the cell (see Fig. 7.3). Extraction is deterministic so each cell produces a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

single measurement per metric. Table 7.12 reports the per-cell scores and Table 7.13 the aggregates by framework and by family.

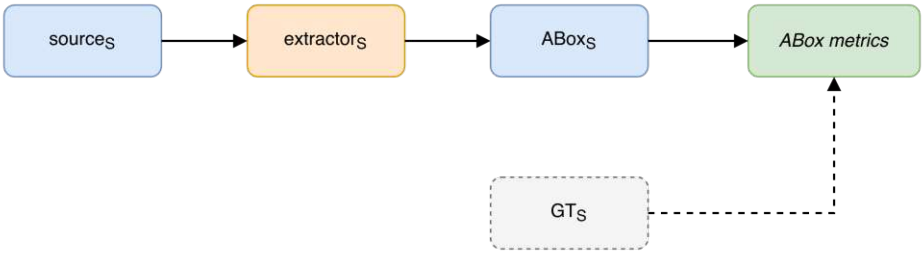

Figure 7.3: Extraction experiment.

Metrics. All four scores use the same precision/recall shape, computed over ABox triples after TBox, ontology-level and blank-node triples are removed. Let G^R and G^C denote the reference and candidate ABox, let P and R denote precision and recall and $F_1 = \frac{2PR}{P+R}$. Every score lies in $[0, 1]$, where 1 means a perfect match between candidate and reference and 0 means no overlap at all.

- Triple F1 compares the two ABoxes as sets of (subject, predicate, object) triples over local names. With T^R and T^C the triple sets of G^R and G^C,

$$P = \frac{|T^R \cap T^C|}{|T^C|}, \qquad R = \frac{|T^R \cap T^C|}{|T^R|}.$$

  It is the strictest of the four and falls whenever the two sides mint different IRIs for the same individual.

- Aligned triple F1 controls for IRI naming. Each candidate individual is matched to a reference individual by a greedy bipartite alignment that maximises the Jaccard similarity of their feature sets, where the feature set is the collection of (predicate, object signature) pairs the individual participates in. After alignment, candidate IRIs are rewritten to their matched reference IRIs and the triple-F1 formula above is recomputed on the rewritten graph. The score reads as a structural similarity estimate that accounts for naming noise.

- Individual F1 groups named individuals by class and compares the per-class counts on each side. Writing n_c^R and n_c^C for the number of individuals of class c in G^R and G^C, class c contributes $\min(n_c^R, n_c^C)$ matches and

$$P = \frac{\sum_c \min(n_c^R, n_c^C)}{\sum_c n_c^C}, \qquad R = \frac{\sum_c \min(n_c^R, n_c^C)}{\sum_c n_c^R}.$$

  The score measures whether each kind of individual appears the right number of times and is insensitive to local names.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- Property F1 is the predicate-level analogue of individual F1: instead of grouping individuals by class, it groups triples by ontology predicate and compares how often each predicate is used. Writing u_p^R and u_p^C for the number of times predicate p is used in G^R and G^C, predicate p contributes $\min(u_p^R, u_p^C)$ matches and

$$P = \frac{\sum_p \min(u_p^R, u_p^C)}{\sum_p u_p^C}, \qquad R = \frac{\sum_p \min(u_p^R, u_p^C)}{\sum_p u_p^R}.$$

  Each relation is therefore credited in proportion to how often it is correctly exercised, not only whether it appears at all.

Results. The extraction experiments (see Tab. 7.12 and 7.13) are consistent across the evaluation corpus. The overall mean Triple F1 score across the evaluation corpus of 0.986 shows that the deterministic pipeline almost always produces the intended output of the ground-truth. The slightly lower aligned F1 score can be explained by the greedy Jaccard matching, as this can produce false-positive matches that are then incorrectly aligned. The likewise very high Individual and Property F1 scores confirm this picture and show that the pipeline can generate a correct semantic representation from source code inputs across different frameworks.

The per-framework and per-family means show near-uniform performance with only minor divergence. LangGraph sits slightly below CrewAI and AutoGen on Triple F1 at 0.976. These differences are small and the Individual and Property F1 scores remain at or close to 1.000 throughout. This shows that AgentOSCIN can faithfully represent agentic AI systems from different frameworks.

Family	Framework	Triple	Aligned	Ind.	Prop.
joke	CrewAI	0.995	0.995	1.000	1.000
	LangGraph	0.991	0.991	1.000	1.000
	AutoGen	0.988	0.988	1.000	1.000
code review	CrewAI	0.997	0.940	1.000	0.996
	LangGraph	0.996	0.996	1.000	0.995
	AutoGen	0.994	0.994	1.000	0.993
tech blog	CrewAI	1.000	0.934	1.000	1.000
	LangGraph	0.996	0.971	1.000	0.995
	AutoGen	0.988	0.988	1.000	1.000
meeting assistant	CrewAI	1.000	1.000	1.000	1.000
	LangGraph	0.957	0.957	1.000	0.991
	AutoGen	0.983	0.983	1.000	1.000
travel planning	CrewAI	1.000	0.947	1.000	1.000
	LangGraph	0.950	0.939	1.000	0.996
	AutoGen	0.966	0.949	1.000	1.000
maths	CrewAI	1.000	1.000	1.000	1.000
	LangGraph	0.967	0.967	0.973	0.982
	AutoGen	0.985	0.985	1.000	1.000
Mean		**0.986**	**0.974**	**0.998**	**0.997**

Table 7.12: Per-cell extraction metrics.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Aggregate	Triple	Aligned	Ind.	Prop.
CrewAI	0.999	0.969	1.000	0.999
LangGraph	0.976	0.970	0.995	0.993
AutoGen	0.984	0.981	1.000	0.999
joke	0.991	0.991	1.000	1.000
code review	0.996	0.977	1.000	0.995
tech blog	0.995	0.964	1.000	0.998
meeting assistant	0.980	0.980	1.000	0.997
travel planning	0.972	0.945	1.000	0.999
maths	0.984	0.984	0.991	0.994

Table 7.13: Per-framework and per-family means of the extraction metrics.

7.2.2 LLM-based Extraction

Experiment Setup. The LLM extraction baseline replaces the deterministic extractor with a single `gpt-5-mini` call. The LLM is prompted with the ontology schema and the source code and is asked to output a populated ontology. This baseline uses the same setup (prompt and pipeline) as AgentO [EKEK26]. The output is parsed with a small repair step that adds missing standard prefixes and removes any prose preceding the first `@prefix` directive.

Metrics. The baseline is reported using the same ABox metric suite as the deterministic extractor to provide a reference for what a language model produces without any additional framework knowledge.

Results. The LLM-based approach shows very poor performance in this evaluation setup. The Triple F1 score of almost zero across all extractions was expected, since this metric only rewards exact matches that the LLM was very unlikely to reproduce. The aligned numbers give a fairer comparison point. The overall mean across all successful runs is 0.339, which indicates poor performance at extracting the intended semantic representation of the agentic AI systems and is notably lower than the deterministic approach. These low scores can also be attributed to the evaluation score setup. While the LLM produces viable semantic representations of the given source code it struggles with the exact conventions defined for the semantic representation. The repair step tries to align the output and the ground-truths before evaluation but the non-deterministic nature of the LLM output made this step difficult. The low alignment metrics need to be interpreted against these evaluation constraints.

Individual and Property F1 show a better picture, with the model extracting the correct classes more reliably (0.697) than the correct properties (0.522), which suggests it identifies what components are present but struggles to connect the relations between them correctly. The aggregate view per framework and per family does not show notably different results. The marginally better scores for CrewAI can be attributed to the framework offering a more natural and declarative "language" that is easier for an LLM to interpret than the graph and node wiring of LangGraph or the group-chat assembly of AutoGen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

One cell failed to produce parseable Turtle, the LangGraph meeting assistant case, so the means are computed over the 17 remaining cells.

Family	Framework	Triple	Aligned	Ind.	Prop.
joke	CrewAI	0.000	0.402	0.835	0.617
	LangGraph	0.000	0.307	0.640	0.274
	AutoGen	0.000	0.341	0.683	0.511
code review	CrewAI	0.159	0.470	0.735	0.556
	LangGraph	0.000	0.301	0.655	0.596
	AutoGen	0.000	0.315	0.683	0.551
tech blog	CrewAI	0.131	0.352	0.706	0.612
	LangGraph	0.000	0.341	0.635	0.536
	AutoGen	0.000	0.344	0.636	0.524
meeting assistant	CrewAI	0.000	0.265	0.525	0.388
	LangGraph (parse failed)				
	AutoGen	0.000	0.278	0.743	0.500
travel planning	CrewAI	0.000	0.461	0.883	0.629
	LangGraph	0.000	0.367	0.716	0.554
	AutoGen	0.000	0.375	0.772	0.549
maths	CrewAI	0.000	0.356	0.737	0.637
	LangGraph	0.000	0.223	0.622	0.455
	AutoGen	0.000	0.270	0.649	0.386
Mean		**0.017**	**0.339**	**0.697**	**0.522**

Table 7.14: Per-cell LLM-baseline extraction metrics (gpt-5-mini).

Aggregate	Triple	Aligned	Ind.	Prop.
CrewAI	0.048	0.384	0.737	0.573
LangGraph	0.000	0.308	0.654	0.483
AutoGen	0.000	0.321	0.694	0.503
joke	0.000	0.350	0.719	0.467
code review	0.053	0.362	0.691	0.567
tech blog	0.044	0.346	0.659	0.557
meeting assistant	0.000	0.272	0.634	0.444
travel planning	0.000	0.401	0.790	0.577
maths	0.000	0.283	0.669	0.492

Table 7.15: Per-framework and per-family means of the LLM-baseline extraction metrics (gpt-5-mini).

7.2.3 Generation

Experiment Setup. The generation experiment runs the generator on each cell's hand-authored ABox and compares the resulting source tree against the cell's original source code (see Fig. 7.4). Table 7.16 reports the per-cell scores and Table 7.17 the per-framework and per-family aggregates of the AST diff.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

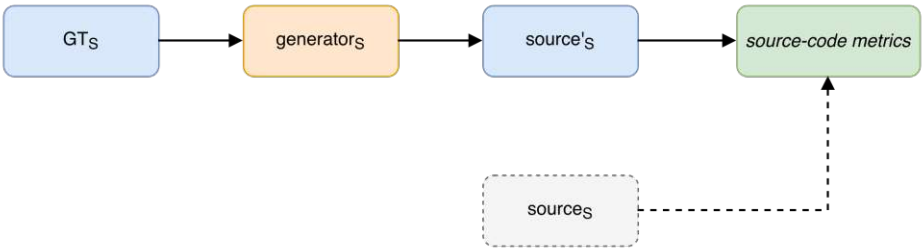

Figure 7.4: Generation experiment.

Metrics. The evaluation setup for the generation experiments covers only the structural metrics. They only indicate how closely the structure of the generated code matches the expected code, but they say nothing about whether the code actually produces the expected outputs.

- Syntax validity reports the parse rate as the fraction of files that parse cleanly under the Python `ast` module without intervention.
- AST diff extracts from each Python file nine sets of structural features: imported top-level modules, class names, class-and-bases signatures, function names, decorator string-arguments such as `@listen("x")`, state-class field names, state-class field annotations such as `Annotated[..., add_messages]` and graph-construction calls such as `add_node` or `add_edge`. Features are unioned across all files in the cell and for each feature class precision and recall are computed against the reference set. The overall score is the macro average across the nine classes, reported as $\bar{F}_1$.

Results. The source code generation metrics are high but of limited informative value. The parse rate is 1.000 for every generated cell, which means the pipeline always produces syntactically valid Python that produces a well-formed AST. Parsing is a minimal requirement and says little about quality, as this metric says nothing about whether the generated structure is correct. The AST F1 scores, with an overall mean of 0.882, are the more telling structural measure. Evaluated against the structural feature set defined in the metrics, they show that the generated source code consistently matches the structure of the expected output in the ground truth across all families and frameworks. These results show that the transformation pipeline can produce structurally correct source code, but they do not tell anything about whether the resulting agentic AI system actually runs and produces correct and expected outputs. This will be tested in the runtime executability experiment.

Looking at the aggregates, AST F1 is consistent across the three frameworks, with CrewAI at 0.907 and AutoGen at 0.908 and LangGraph trailing slightly at 0.831. The lower LangGraph score reflects the additional structural detail of its state-graph and node wiring, which leaves more room for the generated structure to diverge from the ground-truth. Across the task families the spread is wider. The tech blog and travel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

planning families generate most faithfully at 0.930 and 0.915, while the meeting assistant family is the weakest at 0.785, which suggests its richer coordination structure is harder to reconstruct exactly.

Family	Framework	Parse	AST F1
joke	CrewAI	1.000	0.984
	LangGraph	1.000	0.733
	AutoGen	1.000	0.900
code review	CrewAI	1.000	0.894
	LangGraph	1.000	0.896
	AutoGen	1.000	0.928
tech blog	CrewAI	1.000	0.984
	LangGraph	1.000	0.906
	AutoGen	1.000	0.900
meeting assistant	CrewAI	1.000	0.768
	LangGraph	1.000	0.729
	AutoGen	1.000	0.857
travel planning	CrewAI	1.000	0.984
	LangGraph	1.000	0.815
	AutoGen	1.000	0.945
maths	CrewAI	1.000	0.826
	LangGraph	1.000	0.907
	AutoGen	1.000	0.916
Mean		**1.000**	**0.882**

Table 7.16: Per-cell generation metrics. Parse rate and AST F1.

Aggregate	Parse	AST F1
CrewAI	1.000	0.907
LangGraph	1.000	0.831
AutoGen	1.000	0.908
joke	1.000	0.872
code review	1.000	0.906
tech blog	1.000	0.930
meeting assistant	1.000	0.785
travel planning	1.000	0.915
maths	1.000	0.883

Table 7.17: Per-framework and per-family means of the generation metrics.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

7.2.4 Cross-Framework Translation

Experiment Setup. The cross-framework experiment feeds the hand-authored ABox of a family in framework S to the generator with target framework $T \neq S$, feeds the generated code back to the extractor for T, and compares the re-extracted ABox against the hand-authored target ABox for T. The generated code is also compared against the hand-authored target source code with the source code metrics (see Fig. 7.5).

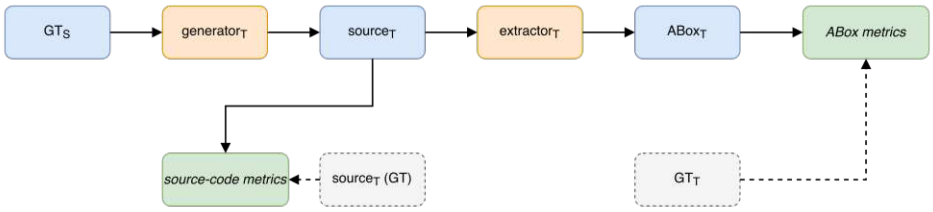

Figure 7.5: Cross-framework round-trip experiment.

Metrics. The cross-framework experiments use both metrics suites (i.e. extraction metrics and generation metrics), as both directions are compared in this setup.

Results. The cross-framework experiments show mixed results. The mean Triple F1 of 0.360 indicates a substantial loss of information at the level of exact triple matches, whereas the aligned score of 0.569 shows that a lenient measure still preserves around 57% of the intended knowledge graph. The structural metrics show the same pattern. Overall AST F1 drops from 0.882 in the same-family generation experiments to 0.596 here. Individual and Property F1 still remain high at 0.765 and 0.731 respectively, which shows that the pipeline preserves classes and properties reliably but struggles to reconnect them correctly.

The averages by direction are more informative. AutoGen as a target shows the strongest scores across the board, while AutoGen as a source produces the weakest scores on the aligned, individual, property and AST metrics. LangGraph and CrewAI settle in between as both source and target. This aligns with our findings in the requirements elicitation 4, where AutoGen is the framework that does not provide complex flow or state-graph structures but instead relies on simple agent definitions wired together through group chats. Translating into AutoGen therefore discards comparatively little structure, since the target representation is shallow, whereas translating out of AutoGen leaves the pipeline with little structural detail to recover. This drops down the metrics in that direction.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Family	Direction	Triple	Aligned	Ind.	Prop.	AST
joke	CrewAI→LangGraph	0.357	0.546	0.750	0.681	0.610
	CrewAI→AutoGen	0.781	0.781	0.923	0.937	0.900
	LangGraph→CrewAI	0.111	0.459	0.636	0.624	0.724
	LangGraph→AutoGen	0.329	0.785	0.973	0.958	0.900
	AutoGen→CrewAI	0.319	0.412	0.525	0.567	0.415
	AutoGen→LangGraph	0.259	0.317	0.444	0.337	0.371
code rev.	CrewAI→LangGraph	0.230	0.526	0.755	0.649	0.482
	CrewAI→AutoGen	0.839	0.793	0.974	0.977	0.928
	LangGraph→CrewAI	0.288	0.568	0.906	0.885	0.538
	LangGraph→AutoGen	0.329	0.720	0.947	0.919	0.667
	AutoGen→CrewAI	0.381	0.458	0.667	0.667	0.544
	AutoGen→LangGraph	0.224	0.306	0.476	0.349	0.309
tech blog	CrewAI→LangGraph	0.257	0.513	0.766	0.623	0.583
	CrewAI→AutoGen	0.323	0.770	0.973	0.984	0.900
	LangGraph→CrewAI	0.260	0.549	0.867	0.874	0.654
	LangGraph→AutoGen	0.356	0.726	0.882	0.873	0.900
	AutoGen→CrewAI	0.377	0.439	0.653	0.640	0.446
	AutoGen→LangGraph	0.257	0.311	0.444	0.309	0.309
meeting	CrewAI→LangGraph	0.240	0.533	0.772	0.699	0.576
	CrewAI→AutoGen	0.485	0.738	0.889	0.865	0.584
	LangGraph→CrewAI	0.092	0.713	0.913	0.921	0.558
	LangGraph→AutoGen	0.529	0.765	0.889	0.849	0.584
	AutoGen→CrewAI	0.211	0.423	0.541	0.563	0.394
	AutoGen→LangGraph	0.311	0.426	0.588	0.535	0.271
travel	CrewAI→LangGraph	0.195	0.424	0.667	0.508	0.413
	CrewAI→AutoGen	0.160	0.629	0.894	0.878	0.945
	LangGraph→CrewAI	0.298	0.558	0.909	0.898	0.586
	LangGraph→AutoGen	0.622	0.679	0.894	0.875	0.945
	AutoGen→CrewAI	0.394	0.467	0.644	0.610	0.447
	AutoGen→LangGraph	0.193	0.234	0.348	0.228	0.270
maths	CrewAI→LangGraph	0.380	0.626	0.837	0.847	0.522
	CrewAI→AutoGen	0.516	0.734	0.968	0.968	0.916
	LangGraph→CrewAI	0.354	0.620	0.789	0.814	0.369
	LangGraph→AutoGen	0.850	0.850	0.968	0.979	0.916
	AutoGen→CrewAI	0.342	0.513	0.703	0.726	0.376
	AutoGen→LangGraph	0.500	0.565	0.750	0.711	0.595
Mean		**0.360**	**0.569**	**0.765**	**0.731**	**0.596**

Table 7.18: Per-cell cross-framework translation metrics.

Direction	Triple	Aligned	Ind.	Prop.	AST
CrewAI → AutoGen	0.517	0.741	0.937	0.935	0.862
LangGraph → AutoGen	0.503	0.754	0.925	0.909	0.819
CrewAI → LangGraph	0.276	0.528	0.758	0.668	0.531
LangGraph → CrewAI	0.234	0.578	0.837	0.836	0.572
AutoGen → CrewAI	0.337	0.452	0.622	0.629	0.437
AutoGen → LangGraph	0.291	0.360	0.509	0.412	0.354
Mean	**0.360**	**0.569**	**0.765**	**0.731**	**0.596**

Table 7.19: Cross-framework translation metrics by ordered direction, averaged over the six task families.

7.2.5 Runtime Executability

Experiment Setup The preceding generation metrics show structural correctness but not that a generated system runs and does what its task requires. We therefore execute every generated artefact and evaluate goal attainment. Each of the 18 generation cells

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

(regenerate framework F from the ontology of a system originally written in F) and 36 cross-framework cells (translate from source S to target T, the six ordered framework pairs over six task families) is run in an isolated subprocess via `runpy`, with the project environment loaded so the agents have model access, the target directory on the import path and a 90 s wall-clock budget. Because LangGraph stores results in graph state, its entry point is instrumented to print the full final state, so every framework's product is observable.

Each non-goal cell is then passed through a repair model (`gpt-5-mini`) once. The prompt provides the full original source of the reference system (code, agent/crew definitions, configuration, tools), the source ontology in Turtle as the semantic specification, the broken generated code with a manifest of its files and the observed runtime failure, under the constraint to keep the target framework, the file layout and the tool-import structure, editing only node/agent bodies, prompt wiring, state access and graph/flow edges. The experiment is single shot (one repair attempt per cell).

Metrics. Goal attainment is assessed against each family's intended output (e.g. a joke, a code review naming a concrete finding, a blog article, one or more action items extracted from a transcript, a travel itinerary, or the arithmetic answer). The maths family is verified deterministically (the output must contain the correct result, $(40+12)\times 6 = 312$). The open-ended families are judged by manual inspection of the produced output against the goal. Importantly, the artefact must be produced for the specified task input: a system that runs to completion but silently substitutes an input of its own invention (e.g. an itinerary for a destination nobody requested) does not reach the goal. Each run is assigned one of three outcomes:

- ✓ **Goal reached**: the system runs to completion and produces the intended artefact for the specified task input.
- ∼ **Partial**: the system runs but does not fully reach the goal: it either produces part of the goal (e.g. an audit without the summary, an itinerary without the final-plan field, a transcript pipeline that extracts zero tasks) or runs without producing it (empty wrapper bodies, or un-wired input). This includes runs where the model, finding no input, fabricates its own and produces a plausible but unrequested artefact.
- ✗ **Error**: the system does not run to completion (e.g. a runtime exception, or a timeout).

Every non-✓ cell is additionally labelled with one issue category naming the dominant cause, defined in Table 7.20.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Category	Meaning	Typical fix
Input wiring	Runs and calls the model but the task input is never interpolated into the prompt. The agent then asks for or echoes the missing input, or invents one and produces an artefact for a task that was never specified.	Interpolate the real input into the prompt (replace the literal placeholder with the state field, e.g. `f"...{state['question']}"`) and seed the entry point with a concrete value.
Stub bodies	The orchestration wrapper (CrewAI Flow or LangGraph subgraph) runs empty or placeholder node bodies so no agent work happens.	Replace the empty or `TODO` body with a real model-invoking call (`model.invoke(...)` / `crew.kickoff(...)`) and return the produced value into state.
Partial output	Runs and produces part of the goal but leaves one node or state field unpopulated.	Implement the missing node, or map the model response to the correct state key so the final field is populated and printed.
Control flow	A graph or router defect, such as an edge to a node that does not exist or a router returning a key that is absent from the edge mapping.	Align the router's return labels with the edge-mapping keys and point every edge at a declared node (add the missing node or correct the target).
Convention mismatch	Framework-specific API misuse, for example reading a CrewAI Pydantic state with the LangGraph dict convention, calling a LangChain `StructuredTool` object directly, or assigning to `StateGraph.state`.	Use the framework-correct construct: Pydantic `state.field` versus dict `state["field"]`, invoke a tool via `tool.invoke(args)` and return state deltas rather than mutating `StateGraph.state`.
Missing symbol	A referenced name or module is undefined or not importable (`NameError` or `ModuleNotFoundError`).	Add the missing import or definition (or install the package). For intra-package modules, fix the import path or package layout.
Schema	A framework data-model or validation constraint is violated, for example a CrewAI Flow state model that requires an `id` field, an `Agent` or `Task` config that fails Pydantic validation, or an AutoGen `FunctionTool` that requires type-annotated tool parameters.	Add the required field(s) to the state model so it satisfies the framework's constraint (e.g. declare `id` on the Flow state).
Timeout	The system does not terminate within the 90 s budget, typically an unbounded agent loop.	Bound the loop: a termination condition or `max_turns`, a recursion or step limit, or a router exit branch that is actually reachable.
Runtime	A failure raised while the agent or model call executes that is not a structural defect. The invocation itself fails, the model output cannot be parsed, or a message-type mismatch occurs at execution (e.g. `No AIMessage in input` or `tuple has no .content`).	Harden the model interaction: pass the message type the framework expects, validate the model output defensively and surface or retry on transient invocation errors.

Table 7.20: Issue categories assigned to every non-goal-reached runtime outcome.

Results Before repair, only 3 of 54 generated systems fully reach their goal (3/18 generation, 0/36 cross-framework; Table 7.21 and Table 7.23). Several further systems run to completion and produce a plausible artefact, but for an input the model invented itself. This is because the concrete task input is never wired in. When calling `kickoff()`, it leaves the literal `{topic}`/`{request}` placeholder in the prompt, or a dummy seed (`"sample topic"`) replaces the real value. The model then picks its own topic, code or destination (e.g. the regenerated CrewAI travel planner, for instance, produced a polished itinerary for Kyoto although the reference request asks for a 10-day trip to Luxembourg). We score these as input-wiring partials, not goal attainment.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The three genuine successes are all AutoGen regenerations. This can be explained as AutoGen embeds the concrete input in the `task=` string of `run_stream(...)`, which the ontology captures as the task description, whereas CrewAI (`kickoff(inputs={...})`) and LangGraph (the initial-state seed) bind the input at run time, outside the ontology, so the deterministic generator cannot restore it.

Overall the non-goal cells are dominated by stub bodies and input wiring: 44 cells run but produce incorrect outputs. A single `gpt-5-mini` repair pass, given the original source, the ontology and the broken skeleton and constrained to produce genuine model-driven output (no hardcoded or framework-bypassing solutions), verified by re-running each system, raises goal attainment to 29 of 54 (54 %)): 14/18 generation and 15/36 cross-framework (Table 7.22 and Table 7.24). The "runs-but-no-goal" category collapses from 44 cells to one, confirming that the residual gap the deterministic generator leaves (i.e. un-synthesised node/crew bodies and un-interpolated or un-seeded inputs) is local and fillable once the structure and grounding are already correct. The cells the repair does not fix surface as genuine framework-specific errors: 24 in total, dominated by API/convention mismatches (e.g. calling a LangChain `StructuredTool` or `ChatOpenAI` object directly, or assigning to `StateGraph.state`) and agent-schema validation failures (i.e. faults that require deeper, framework-specific synthesis than a single pass provides).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Family	Framework		Category	Observed behaviour
joke	CrewAI	∼	Input wiring	`{topic}` unfilled; invents own topic
joke	LangGraph	∼	Input wiring	Asks for a topic; joke fields placeholder
joke	AutoGen	✓	—	Joke
code-review	CrewAI	∼	Input wiring	`{code}` unfilled; reviews no real code
code-review	LangGraph	∼	Input wiring	Dummy "sample code" seed; summary node unwired
code-review	AutoGen	∼	Input wiring	Prompt not interpolated; asks for code
tech-blog	CrewAI	∼	Input wiring	`{topic}` unfilled; invents own topic
tech-blog	LangGraph	∼	Input wiring	Dummy "sample topic" seed; invents topic
tech-blog	AutoGen	✓	—	Article
meeting	CrewAI	✗	Missing symbol	`NameError`: MeetingTask undefined
meeting	LangGraph	∼	Input wiring	Reads seed message, asks for transcript
meeting	AutoGen	∼	Input wiring	Literal transcript placeholder
travel	CrewAI	∼	Input wiring	`{request}` unfilled; invents Kyoto trip
travel	LangGraph	∼	Input wiring	Dummy "sample request" seed; final-plan node unwired
travel	AutoGen	✗	Timeout	Unbounded agent loop (>90 s)
maths	CrewAI	∼	Input wiring	Asks for the query
maths	LangGraph	✗	Control flow	`KeyError`: None (router stub)
maths	AutoGen	✓	—	Computes 312
✓ 3	∼ 12	✗ 3		

Table 7.21: Generation runtime outcomes, before fixup. ✓ goal reached, ∼ partial / runs but goal not reached, ✗ error.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Family	Framework		Category	Observed behaviour
joke	CrewAI	✓	—	Joke (about the seeded topic)
joke	LangGraph	✓	—	Joke
joke	AutoGen	✓	—	Joke
code-review	CrewAI	✓	—	Review with CWE findings
code-review	LangGraph	✓	—	Review with findings
code-review	AutoGen	✓	—	Review (flags `eval`)
tech-blog	CrewAI	✓	—	Article
tech-blog	LangGraph	✓	—	Article
tech-blog	AutoGen	✓	—	Article
meeting	CrewAI	✗	Schema	`ValidationError`: Flow state needs 'id'
meeting	LangGraph	∼	Partial output	Tasks in reply, but not mapped into state; CSV empty
meeting	AutoGen	✗	Schema	`KeyError`: 'tasks' (unannotated tool param)
travel	CrewAI	✓	—	Itinerary (Luxembourg)
travel	LangGraph	✓	—	Itinerary (Luxembourg)
travel	AutoGen	✓	—	Itinerary
maths	CrewAI	✗	Schema	`Task/Agent` config validation fails
maths	LangGraph	✓	—	Answer 312
maths	AutoGen	✓	—	Answer 312
✓ 14	∼ 1	✗ 3		

Table 7.22: Generation runtime outcomes, after one `gpt-5-mini` repair pass.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Family	Dir.		Category	Observed behaviour
joke	C→L	✗	Control flow	Edge to unknown node Fail
joke	C→A	∼	Input wiring	Literal `{topic}` in task; invents topic
joke	L→C	✗	Convention mismatch	`StateWithId` not subscriptable
joke	L→A	∼	Input wiring	Vague task string; invents own joke
joke	A→C	∼	Stub bodies	Flow runs, crew empty
joke	A→L	∼	Stub bodies	Subgraph node stub
code-review	C→L	∼	Stub bodies	Placeholder pass-through
code-review	C→A	∼	Input wiring	Review template; asks for code
code-review	L→C	∼	Stub bodies	Flow methods empty
code-review	L→A	∼	Input wiring	Prompt not interpolated
code-review	A→C	∼	Stub bodies	Flow runs, crew empty
code-review	A→L	∼	Stub bodies	Subgraph node stub
tech-blog	C→L	∼	Stub bodies	Placeholder pass-through
tech-blog	C→A	∼	Input wiring	Literal `{topic}` in task; invents topic
tech-blog	L→C	∼	Stub bodies	Flow methods empty
tech-blog	L→A	∼	Input wiring	Asks for the topic
tech-blog	A→C	∼	Stub bodies	Flow runs, crew empty
tech-blog	A→L	∼	Stub bodies	Subgraph node stub
meeting	C→L	✗	Missing symbol	`NameError`: MeetingTask
meeting	C→A	∼	Input wiring	No transcript content
meeting	L→C	∼	Stub bodies	Flow methods empty
meeting	L→A	∼	Input wiring	Literal transcript placeholder
meeting	A→C	∼	Stub bodies	Flow runs, crew empty
meeting	A→L	∼	Stub bodies	Subgraph node stub
travel	C→L	∼	Stub bodies	Placeholder pass-through
travel	C→A	∼	Input wiring	Literal `{request}`; invents Rome trip
travel	L→C	∼	Stub bodies	Flow methods empty
travel	L→A	∼	Input wiring	Asks for trip details
travel	A→C	∼	Stub bodies	Flow runs, crew empty
travel	A→L	∼	Stub bodies	Subgraph node stub
maths	C→L	∼	Stub bodies	Placeholder pass-through
maths	C→A	∼	Input wiring	Asks for the query
maths	L→C	✗	Convention mismatch	`StateWithId` not subscriptable
maths	L→A	∼	Input wiring	Asks for clarification
maths	A→C	∼	Stub bodies	Flow runs, crew empty
maths	A→L	∼	Stub bodies	Subgraph node stub
✓ 0	∼ 32	✗ 4		

Table 7.23: Cross-framework runtime outcomes, before fixup. Directions: C=CrewAI, L=LangGraph, A=AutoGen (source→target).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Family	Dir.		Category	Observed behaviour
joke	C→L	✗	Control flow	`KeyError`: 'Pass'
joke	C→A	✓	—	Joke (seeded topic)
joke	L→C	✗	Schema	`ValidationError` (Agent)
joke	L→A	✓	—	Joke (seeded topic)
joke	A→C	✗	Convention mismatch	`Crew` has no `.run`
joke	A→L	✓	—	Joke
code-review	C→L	✗	Convention mismatch	`StructuredTool` not callable
code-review	C→A	✓	—	Review
code-review	L→C	✗	Missing symbol	`ModuleNotFound`: tools
code-review	L→A	✓	—	Review
code-review	A→C	✗	Convention mismatch	inputs must be a dict
code-review	A→L	✗	Convention mismatch	`ChatOpenAI` not callable
tech-blog	C→L	✓	—	Article
tech-blog	C→A	✓	—	Article (seeded topic)
tech-blog	L→C	✓	—	Article
tech-blog	L→A	✓	—	Article
tech-blog	A→C	✗	Schema	`ValidationError` (Agent)
tech-blog	A→L	✗	Convention mismatch	`ChatOpenAI` not callable
meeting	C→L	✗	Control flow	`InvalidUpdateError` (tasks)
meeting	C→A	✗	Timeout	timeout (>90 s)
meeting	L→C	✗	Runtime	LLM call failed
meeting	L→A	✓	—	Tasks
meeting	A→C	✗	Runtime	agent invocation failed
meeting	A→L	✗	Runtime	No AIMessage in input
travel	C→L	✓	—	Itinerary
travel	C→A	✓	—	Itinerary (Luxembourg)
travel	L→C	✗	Convention mismatch	`StateGraph.state` has no setter
travel	L→A	✗	Convention mismatch	`SelectorGroupChat` init
travel	A→C	✗	Schema	`ValidationError` (Agent)
travel	A→L	✗	Runtime	model output unparsable
maths	C→L	✓	—	Answer 312
maths	C→A	✓	—	Answer 312
maths	L→C	✗	Convention mismatch	`APIRemovedInV1` (deprecated call)
maths	L→A	✓	—	Answer 312
maths	A→C	✗	Schema	`ValidationError` (Agent)
maths	A→L	✗	Runtime	tuple has no `.content`
✓ 15	∼ 0	✗ 21		

Table 7.24: Cross-framework runtime outcomes, after one `gpt-5-mini` repair pass. Directions: C=CrewAI, L=LangGraph, A=AutoGen (source→target).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 8

Result Analysis

This chapter first interprets the evaluation against the four research questions the thesis set out to answer. RQ1 asks what an interoperability ontology must represent. RQ2 asks whether AgentOSCIN satisfies those requirements as a representation. RQ3 and RQ4 ask to what extent the AgenticBIT transformation framework produces correct bidirectional transformations between source code and ontology instances. Additionally, this chapter also provides the main limitations that come with the evaluation setup.

8.1 Answers to Research Questions

This section answers each research question in turn, drawing the evidence together from the preceding chapters. RQ1 is answered by the requirements elicitation of Chapter 4, which establishes what a framework-agnostic ontology must represent. RQ2 is answered by the CQ-questions and structural ontology evaluation, which assess whether AgentOSCIN satisfies those requirements as a representation. RQ3 and RQ4 are answered by the extraction and generation experiments respectively, which measure how faithfully the AgenticBIT pipeline transforms between source code and ontology instances in each direction.

RQ1: What are the requirements for an ontology to facilitate interoperability between modern agentic AI frameworks? This question is answered by the requirement elicitation, which produced two results that together define what a framework-agnostic ontology must represent. The concept matrix in Table 4.1 maps the literature corpus onto seven recurring concept areas (i.e. Agent definition, perception and task, coordination, reasoning, memory, tools and guardrails/governance). These areas emerge from the literature corpus. The framework analysis of CrewAI, AutoGen and LangGraph then refines them into the twenty-one representational requirements in Table 4.2, each

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

stated as a concrete capability and each traced to the literature concept, the framework construct or both that motivate it.

RQ2: To what extent does the proposed ontology satisfy the requirements for agentic AI interoperability? The competency-question evaluation is the most direct measure of whether the representation meets the requirements. When run over the populated knowledge graph, all thirty-five competency questions resolve to full coverage (Table 7.10), against eight full for AgentO and six full for the DBM model. Expressed as a rate this is a rise from 22.9 % full coverage under AgentO and 17.1 % under DBM to 100 % under the extension. The extension is therefore the part that makes the representation usable for this purpose, as it introduces the concepts a description of an agentic system needs, which the base vocabulary was missing.

The per-question breakdown in Table 7.9 shows that the two baselines fail in complementary ways. The DBM model [DBM25] contributes task dependencies, task-level tool binding and guardrail validation that AgentO [EKEK26] cannot express (CQ-14, CQ-33, CQ-34), but it loses the agent identity, prompt-component structure and team membership that AgentO does carry (CQ-2, CQ-6, CQ-19). A representation built from either baseline alone would leave one half of the requirement set unanswered. More importantly, both baselines leave the same block of agentic concepts entirely uncovered: coordination patterns, termination conditions, conditional workflow branches, reasoning patterns and human-in-the-loop checkpoints (CQ-18, CQ-21, CQ-24, CQ-25 to CQ-27 and CQ-35). Their full coverage under AgentOSCIN and absence under both baselines shows the importance of the extended ontology.

Structural quality supports the same conclusion. The OOPS! scan reported no critical pitfalls, and the important and minor pitfalls that remained either originate in the reused vocabularies or were corrected directly, which leaves the AgentOSCIN extensions free of structural defects. The vocabulary that delivers the coverage is therefore not achieving it through modelling shortcuts that a pitfall scanner would flag.

RQ3: To what extent can an ontological representation be systematically extracted from an agentic AI system's source code?

Extraction Within a Framework Against the hand-authored ground-truth ABoxes the extractor reaches a mean Triple F1 of 0.986 and an Aligned F1 of 0.974, with Individual F1 at 0.998 and Property F1 at 0.997. It recovers the intended vocabulary and the intended set of entities essentially without loss.

How easily a framework yields its structure tracks how declaratively it states it. CrewAI configures agents and tasks in YAML, so the information already sits in nearly the shape the ontology wants, and it scores highest (Triple F1 0.999). LangGraph sits at the other end (0.976), with no real declarative surface and much of its structure buried in imperative calls such as `add_node` and `add_edge`. AutoGen falls between the two (0.984).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The LLM baseline is where a deterministic extractor shows its strengths. Given the full ontology and a current reasoning model it still reached an Aligned F1 of only 0.339 and a Triple F1 of essentially zero, generating its own identifiers on almost every entity and failing to produce parseable Turtle on one of the eighteen cells. The difference lies in where each approach stores its framework-specific knowledge. The pipeline encodes it once, in explicit mappings, and applies it identically every time, while the LLM must be told it in the prompt and even then paraphrases, hallucinates properties the schema never defines. Pushing that mapping knowledge into a prompt would also bloat it and introduce its own failure modes, which is what we saw.

Extraction Across a Framework Boundary Across framework boundaries the numbers are noticeably worse. The extractor now runs not on clean human-authored source code but on the code the generator produces for a target framework, scored against that target's own hand-authored ground-truth. The mean Triple F1 falls to 0.360 and the Aligned F1 to 0.569.

Triple F1 is hard to read as extraction quality here, as the strict metric punishes every renamed individual even when the entity behind it is correct. The aligned score is the fairer metric. At 0.569, far below the within-framework 0.974, it shows that even once naming is discounted a real share of the intended triples does not survive the round trip. The extractor recovers only what the generated code still contains.

The class and predicate metrics explain that loss more precisely. Individual F1 and Property F1 are still high with an F1 at 0.765 and 0.731. The system still has the correct classes and properties in roughly the right numbers. The links between them are lost. The pipeline keeps that a team has agents, tasks and tools of the expected classes while dropping a portion of the specific edges that tie the classes together.

Because the extractor reads the generated target code, two things come together. First, how declarative the target exposes its structure for re-extraction and second, how much structure the source carried into that code to begin with. AutoGen is the central point for both directions. Generating into AutoGen extracts almost as cleanly as within a single framework (CrewAI→AutoGen and LangGraph→AutoGen reach Aligned F1 of 0.741 and 0.754 with Individual F1 above 0.92), since the target is flat and declarative. Generating out of AutoGen is weakest (AutoGen→CrewAI = 0.452, AutoGen→LangGraph = 0.360), because a thin AutoGen source carries little structure into the richer target and leaves the extractor little to recover. Extraction is only as good as the structure present in the code it is handed, and across the boundary that structure is set by the source and target conventions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

RQ4: To what extent can an agentic AI system's source code be systematically generated from its ontological representation?

Generation Within a Framework Generation is the harder direction. The results split into structural quality and executability. The structure of the generated code, which is strong and its executability, which is more complicated and depends heavily on the framework.

Structurally, every generated cell parses without intervention and the mean AST diff against the hand-authored source reaches 0.882, so the generated code reproduces most of the constructs of the original. AutoGen works well here, as its architecture is flat and declarative at the API level, where agents are instantiated as objects, tools are wrapped in constructors and a team is assembled in a single call. There is little routing logic to reconstruct, so the generated AutoGen code mirrors the original most closely (AST F1 = 0.908). CrewAI follows closely (0.907) while LangGraph is lowest (0.831), because so much of its behaviour lives in imperative graph construction that a declarative representation under-specifies.

Where generation becomes more demanding is in producing code that not only mirrors the original but actually runs and produces the expected outputs. We therefore executed every generated cell and judged it against the output the system was meant to produce. Before any repair three of the eighteen same-framework cells run to completion and produce the intended artefact (0.167), twelve run but fall short of the goal and three raise errors. The failures concentrate on a small set of recurring causes (see Tab. 7.20). The most common is task input that the entry point never wires in, so the system either asks for it or quietly invents its own and produces a plausible but unrequested artefact. The remainder are harder defects: an undefined symbol, a router returning a key absent from its edge mapping and one unbounded loop that exceeds the time budget.

That the three genuine successes are all AutoGen cells is itself a finding. AutoGen carries the concrete input inside the task string, which the ontology captures, whereas CrewAI and LangGraph bind it at run time, outside anything the generator can restore. A single `gpt-5-mini` repair pass, given the original source, the ontology and the broken skeleton and constrained to produce genuine model-driven output, raises goal attainment to fourteen of eighteen (0.778) and collapses the "runs-but-no-goal" category to a single cell.

Generation Across a Framework Boundary The cross-framework experiment is the most demanding form of generation, because the target framework differs from the one the representation came from.

The most important cross-framework finding is an asymmetry of the examined frameworks. Generating into AutoGen works very well in both directions of travel (CrewAI→AutoGen and LangGraph→AutoGen reach AST F1 of 0.862 and 0.819), because the flat agent-chat target asks for little structure to be invented. Generating out of AutoGen is noticeably

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

lossier (AutoGen→CrewAI = 0.437, AutoGen→LangGraph = 0.354), because AutoGen reifies an entire multi-agent team as a single delegated task and therefore never records the per-step structure that a CrewAI Flow or a LangGraph StateGraph needs in order to be faithfully reconstructed.

The executability results show the same asymmetry. Before repair, none of the thirty-six translations run to completion and produce their goals. Four appear to, but manual inspection shows they produce their artefact for an input the model invented itself, and the rest are dominated by stub bodies and un-interpolated input. One `gpt-5-mini` repair pass lifts this to fifteen of thirty-six (41.7 %).

The repair step does not only change the amount of working cells but also how the remainder fail. Before repair the dominant outcome is silent. Thirty-two cells run (partially) to completion because their stub bodies or input wiring never reach the target framework's API. The repair step fills those stub bodies with real model calls, tool invocations and state access, which raises errors from four to twenty-one. After repair, failures surface as concrete errors instead of passing silently, and where the partial cells showed no error they now produce concrete framework-convention faults (e.g. calling a LangChain `StructuredTool` or `ChatOpenAI` object directly, invoking a `Crew` with an AutoGen-style `.run`, assigning to a `StateGraph.state` that has no setter and failing agent-schema validation).

When looking at the results it shows that ten out of twelve translations into AutoGen are successful after repair. Four out of twelve translations into LangGraph are successful and just one out of twelve translations into CrewAI is successful. A flatter target API leaves less room for misuse. CrewAI is very strict and validates agents, tasks and flow state through Pydantic so that near-miss constructions end as `ValidationErrors` before an agent ever runs.

Grouping the successes by source reveals the same structural asymmetry in reverse. Eight of the twelve come from CrewAI, six from LangGraph and just one from AutoGen. A thin AutoGen source gives the repair model the least structure to work from, and its richest reference (the original source code) ends up demonstrating the conventions of the wrong framework. The residual faults reflect this directly. The ontology carries structure but not framework conventions. The remaining failures stem from exactly that missing convention knowledge.

8.2 Limitations

The results of this thesis should be read against the constraints of the evaluation design and the most consequential of those constraints concerns its independence. The ontology, the extraction and generation pipeline, the hand-authored ground-truth ABoxes and the manual framework ports were all the work of a single author. The reference against which extraction is scored and the reference against which generation is scored were therefore written by the same person who designed the artefact being measured, using the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

same modelling conventions throughout. A high extraction Triple F1 therefore records agreement between the extractor and a ground-truth that already shares its assumptions and not agreement with a target established independently of it. The within-pipeline metrics still have value. They remain a good measure of internal consistency and of how much information survives each transformation step. What they cannot do is stand as evidence of correctness against an external standard. A stronger design would draw its ground-truth from independent annotators blind to the implementation and report the agreement between them.

The competency question evaluation shows a similar problem. The questions were derived from the requirements that drove the design of AgentOSCIN, so the full coverage the ontology achieves confirms only that it satisfies the requirements it was built to satisfy. This is a check of internal consistency between specification and artefact and not an external benchmark of adequacy. The comparison against the two baselines is constrained further by an asymmetry in what each was asked to demonstrate. AgentOSCIN is assessed against a populated knowledge graph, AgentO is assessed on the availability of vocabulary alone and the DBM model is assessed hypothetically, under the assumption that its class diagram had been instantiated faithfully. The coverage figures are best read as the relative expressiveness gain of the extension over the prior vocabularies under these differing standards and not as a precise measurement on a single common scale.

The corpus itself is a further limitation. The evaluation rests on six task families ported across three frameworks for eighteen cells, which is a small sample and the originals are based on canonical examples taken from the frameworks' own repositories. These examples tend toward clear and clean structure. The systems that would stress an interoperability representation most heavily are the ones that are underrepresented here (i.e. deeply nested orchestrations, dense conditional routing and large tool inventories).

Another limitation concerns the non-systematic approach of two LLM-based steps in the evaluation setup (i.e. LLM-based extraction and one-shot repair). The prompts, the information supplied to the model and the choice of model itself were all fixed at their initial configuration. Alternative prompt formulations, richer or more selective context and stronger models might each raise performance. But token budget and time constraints did not allow a full exploration in this regard. The reported figures should therefore be read as the performance of one unoptimised configuration. A tuned setup might score higher and is subject to future work.

A final limitation is one of scope. By design the ontology models the architecture of an agentic system and the configuration that wires it together and it deliberately does not capture the imperative bodies of tool functions or the fine-grained control flow inside node implementations. This boundary is the reason a structurally complete generation can still fail to run, as the runtime experiment shows. This is a deliberate scoping decision, since modelling arbitrary executable behaviour would push the representation toward a general programming-language semantics and away from the interoperability vocabulary it is meant to be.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 9

Conclusion and Outlook

This thesis set out to determine to what extent semantic representations can enable interoperability between modern agentic AI frameworks, a challenge repeatedly identified as central to the field [SRK26, DBM25, EKEK26]. The results show that a shared semantic layer can indeed support such interoperability, but that the extent to which it carries executable behaviour is bounded by what the ontology chooses to model.

A consistent pattern across both transformation directions is that interoperability tracks the declarativity of the frameworks involved. The more declaratively a framework exposes its structure, the more faithfully a semantic representation can be extracted from source code. The same dependency is visible for generation. A rich semantic representation, such as one extracted from CrewAI, produces better generation across all frameworks than a representation extracted from a semantically poorer framework such as AutoGen. This asymmetry centres on AutoGen. Generating into it is comparatively easy, because its flat, declarative target demands little structure to be reconstructed, whereas generating out of it is lossy, because its thin source carries little structure into a richer target.

The evaluation also revealed clear boundaries of deterministic generation. The errors that produced dysfunctional or incomplete source code fall into a set of recurring, well-defined categories. Addressing them deterministically would require either encoding a large number of edge cases into the pipeline or enriching the semantic representation with so much implementation detail that it would undermine the very aim of a framework-agnostic representation. We argue that these failures can be resolved more efficiently by delegating the repair of the generated code to a LLM. This is supported by our single repair pass, which raised the share of same-framework cells that run to their intended goal from 3/18 to 14/18. The same step, however, also exposed an ambiguity. Without a carefully constrained prompt, the model tends to hard-code values that mimic the expected output. This ambiguity points directly toward future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Several directions follow from these findings. First, our results suggest that the most effective system is unlikely to be either fully deterministic or fully LLM-based but rather a combination of the two. The LLM-based repair step showed promising results and reliably fixed simple issues such as isolated syntactic errors or minor structural mismatches. Its performance degraded on more complicated cases that required reasoning across multiple parts of the target implementation or deeper knowledge of the target framework. The LLM-based extraction by contrast performed poorly, which indicates that the deterministic pipeline remains the stronger choice for recovering structured information from source code. Taken together these observations point toward a hybrid design in which deterministic components handle extraction and transformation while LLM-based components handle repair.

Building on this, it would be worth investigating whether an agentic AI system can outperform both the deterministic pipeline proposed here and a fully LLM-based approach such as that by Ekelhart et al. [EKEK26] at the transformation task itself. Instead of a single model performing repair in isolation, such a system could consist of several cooperating agents equipped with dedicated tools, for example our AgenticBIT tool, and with richer framework knowledge. Access to tools and framework-specific context could allow the agents to resolve the more complicated repair cases that a single LLM struggled with and to produce more reliable solutions. Given how promising the LLM-based repair already proved, extending it into a tool augmented agentic setting is a natural next step.

Second, current representations, including ours and prior work such as AgentO, focus almost exclusively on the design of agentic systems and do not capture their behaviour at runtime [EKEK26]. Whether a semantic representation can be extended to model the runtime of agentic AI systems remains an open question, and is a natural complement to calls for decoupling an agent's semantic reasoning from its structural coordination logic [BBP25]. Third, the degree of interoperability that is achievable could be tested more directly by modelling agentic systems natively in AgentOSCIN and then generating correct source code for an arbitrary target framework, rather than only transforming between existing implementations.

To summarise, AgentOSCIN and AgenticBIT produced promising results in both transformation directions. Although the limitations mean that these results must be read with appropriate caution, the proof of concept demonstrates that an agentic system can be transferred from one framework to another with predictable behaviour (i.e. in the sense that the cases in which transfer fails can be clearly categorized). While some translations already produce fully working systems, particularly those generating into AutoGen, the majority still produce broken skeletons. If hybrid deterministic and LLM-based systems can close this gap, a semantic representation could become a practical artefact for genuine cross-framework interoperability and thereby help address the interoperability challenges identified in the current literature. This contribution reaches beyond framework interoperability alone: expressing the design of agentic AI systems in a unified, machine-readable form can enable the formal auditing and governance of these systems [MRV+25] and provide a foundation for the broader vision of a robust Web of Agents [PBPS25].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Overview of Generative AI Tools Used

The deterministic AST extraction and generation pipeline was developed with the assistance of Claude Code (Anthropic, Opus 4.6 model), which was used to implement the AST traversals mapping framework-specific source code to the shared intermediate representation layer. The rule-based and deterministic nature of these transformation mappings allowed for iterative refinement and helped mitigate the inherent non-determinism of LLMs: each mapping rule could be inspected, corrected and re-validated independently. It should be noted that these mappings could in principle have been authored manually; the model served primarily as a tool for automating the transcription of well-defined deterministic rules into executable code. For writing support, Grammarly and Claude Sonnet 4.6 were used for grammar and style checking and improvements. The author takes full responsibility for the accuracy and integrity of all results presented.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1 Conceptual foundations of agentic AI [BKN+25]. 6
2.2 Unified class model showing the core components of agentic AI systems [DBM25] 7

3.1 Methodology Overview 11
3.2 OSCIN Method Overview [dAZS21] 14

5.1 AgentOSCIN Overview: Core classes and properties divided into the concept groups 42
5.2 Agent definition instantiation excerpt. 43
5.3 Perception and task instantiation excerpt. 44
5.4 Coordination instantiation excerpt. 46
5.5 Reasoning instantiation excerpt. 48
5.6 Memory instantiation excerpt. 49
5.7 Tools instantiation excerpt. 50
5.8 Guardrails and governance instantiation excerpt. 51

6.1 OSCIN-based extractor pipeline. 54
6.2 Excerpt of the `ExtractedAgent` dataclass. Each field is named after the ontology concept it denotes and annotated in-source with the property it maps to. 55
6.3 OSCIN-based code-generator pipeline. 57
6.4 Excerpt of the AutoGen Jinja2 template, emitting one `AssistantAgent` per SIR agent. 58

7.1 Results of the OOPS! Pitfall Scanner before the corrections 62
7.2 Results of the OOPS! Pitfall Scanner after the corrections 62
7.3 Extraction experiment. 73
7.4 Generation experiment. 77
7.5 Cross-framework round-trip experiment. 79

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1 Concept Matrix: Core Concepts across the Literature Corpus 22
4.2 Representational Requirements for a Semantic Model of Agentic AI Systems 35

5.1 CQs derived from Requirements Elicitation 38
5.2 Coverage of AgentOSCIN classes by AgentO [EKEK26] and the DBM model [DBM25] (✓ fully covered, ∼ partially covered, × not covered). 39
5.3 Coverage of the core AgentOSCIN properties by AgentO [EKEK26] and by the relations of the DBM model [DBM25] (✓ fully covered, ∼ partially covered, × not covered) 40

7.1 AgentOSCIN ontology structural metrics, as reported by Protégé. 61
7.2 Answer to CQ-2 (excerpt; 18 agents in total across the seven systems). . . 65
7.3 Answer to CQ-13 (excerpt). The strategy distinguishes explicitly assigned CrewAI tasks, topology-determined LangGraph nodes and orchestrator-delegated AutoGen team calls. 65
7.4 Answer to CQ-23 (excerpt). The branch from `generate_joke` is conditional (no fixed `nextStep`); the meeting-assistant flow fans out to three parallel sinks at order 2. 66
7.5 Answer to CQ-26. Reasoning patterns are not surfaced by the six corpus systems and are exercised here through the feature-demo system. 66
7.6 Answer to CQ-28. The `ownerKind` is derived from the class of the binding's owner, distinguishing agent-private from system-wide memory. 67
7.7 Answer to CQ-31 (excerpt; 6 tools in total). Input schemas are stored as serialised JSON schema literals. 67
7.8 Answer to CQ-34. 68
7.9 Per-CQ coverage. ✓ full, ∼ partial, × none. 69
7.10 Competency Coverage across all data models 70
7.11 Original source code examples. 72
7.12 Per-cell extraction metrics. 74
7.13 Per-framework and per-family means of the extraction metrics. 75
7.14 Per-cell LLM-baseline extraction metrics (gpt-5-mini). 76
7.15 Per-framework and per-family means of the LLM-baseline extraction metrics (gpt-5-mini). 76
7.16 Per-cell generation metrics. Parse rate and AST F1. 78

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

7.17 Per-framework and per-family means of the generation metrics. 78
7.18 Per-cell cross-framework translation metrics. 80
7.19 Cross-framework translation metrics by ordered direction, averaged over the six task families. 80
7.20 Issue categories assigned to every non-goal-reached runtime outcome. . . . 82
7.21 Generation runtime outcomes, before fixup. ✓ goal reached, ∼ partial / runs but goal not reached, ✗ error. 84
7.22 Generation runtime outcomes, after one `gpt-5-mini` repair pass. 85
7.23 Cross-framework runtime outcomes, before fixup. Directions: C=CrewAI, L=LangGraph, A=AutoGen (source→target). 86
7.24 Cross-framework runtime outcomes, after one `gpt-5-mini` repair pass. Directions: C=CrewAI, L=LangGraph, A=AutoGen (source→target). . . . 87

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Bibliography

[AAC+25] Berk Atil, Sarp Aykent, Alexa Chittams, Lisheng Fu, Rebecca J. Passonneau, Evan Radcliffe, Guru Rajan Rajagopal, Adam Sloan, Tomasz Tudrej, Ferhan Ture, Zhe Wu, Lixinyu Xu, and Breck Baldwin. Nondeterminism of "deterministic" llm settings, 2025.

[AAD25] Hesham Allam, Ban AlOmar, and Juan M. Dempere. Agentic ai for it and beyond: A qualitative analysis of capabilities, challenges, and governance. *The Artificial Intelligence Business Review*, 2025.

[AKA25] Deepak Acharya, Karthigeyan Kuppan, and Divya B Ashwin. Agentic ai: Autonomous intelligence for complex goals – a comprehensive survey. *IEEE Access*, PP:1–1, 01 2025.

[ARH19] Abdelghany Abdelghany, Nagy Ramadan, and Hesham Hefni. An agile methodology for ontology development. *International Journal of Intelligent Engineering and Systems*, 12:170–181, 04 2019.

[ASS15] Robert Arp, Barry Smith, and Andrew D. Spear. *Building Ontologies with Basic Formal Ontology*. 7 2015.

[BBP25] Uwe M. Borghoff, Paolo Bottoni, and Remo Pareschi. Beyond prompt chaining: The tb-cspn architecture for agentic ai. *Future Internet*, 17(8), 2025.

[BKN+25] Ajay Bandi, Bhavani Kongari, Roshini Naguru, Sahitya Pasnoor, and Sri Vidya Vilipala. The rise of agentic ai: A review of definitions, frameworks, architectures, applications, evaluation metrics, and challenges. *Future Internet*, 17(9), 2025.

[Bot25] V. Botti. Agentic ai and multiagentic: Are we reinventing the wheel?, 2025.

[Cla26] Bryan Clark. What is LangGraph? https://www.ibm.com/think/topics/langgraph, 4 2026. Accessed: 2026-04-13.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[CLH+25] Shuaihang Chen, Yuanxing Liu, Wei Han, Weinan Zhang, and Ting Liu. A survey on llm-based multi-agent system: Recent advances and new frontiers in application, 2025.

[Cre25] CrewAI Inc. CrewAI documentation. `https://docs.crewai.com`, 2025. Accessed: 2026-04-15.

[dAZS21] Camila Zacché de Aguiar, Félix Zanetti, and Vitor E. Silva Souza. Source code interoperability based on ontology. In *Proceedings of the XVII Brazilian Symposium on Information Systems*, SBSI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[DBM25] Hana Derouiche, Zaki Brahmi, and Haithem Mazeni. Agentic ai frameworks: Architectures, protocols, and design challenges, 2025.

[EDS+24] Alexis L. Ellis, B. Dave, H. Salehi, S. Ganapathy, and Cogan Shimizu. EASY-AI: sEmantic And compoSable glYphs for representing ai systems. In *HHAI 2024: Hybrid Human Artificial Intelligence*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2024.

[EEQ22] Jalil ElHassouni and Abderrahim El Qadi. Ontology engineering methodologies: State of the art. In *Proceedings of the 5th International Conference on Big Data and Internet of Things (BDIoT 2021)*, volume 489 of *Lecture Notes in Networks and Systems*, pages 59–72. Springer, Cham, 2022.

[EKEK26] Andreas Ekelhart, Kabul Kurniawan, Fajar J. Ekaputra, and Elmar Kiesling. Agento: An ontology for modeling agentic ai systems. In *The Semantic Web: 23th International Conference, ESWC 2026 (accepted for publication)*, 2026.

[EPK25] Fajar J. Ekaputra, Alexander Prock, and Elmar Kiesling. Towards supporting ai system engineering with an extended boxology notation. In *KG-STAR 2025 Knowledge Graphs for Responsible AI 2025*, volume 4018 of *CEUR Workshop Proceedings*. CEUR Workshop Proceedings, August 2025.

[ESS+17] Fajar Ekaputra, Marta Sabou, Estefanía Serral, Elmar Kiesling, and Stefan Biffl. Ontology-based data integration in multi-disciplinary engineering environments: A review. *Open Journal of Information Systems (OJIS)*, 4:1–26, 10 2017.

[FC18] Hector Florez and Marcelo Castro. Model driven engineering approach to configure software reusable components. In *Proceedings of the 1st International Conference on Applied Informatics (ICAI 2018)*, pages 352–363, November 2018.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[FLGPJ97] Mariano Fernández-López, Asuncion Gomez-Perez, and Natalia Juristo. Methontology: from ontological art towards ontological engineering. *Engineering Workshop on Ontological Engineering (AAAI97)*, 03 1997.

[GB09] Maria Grant and Andrew Booth. A typology of reviews: An analysis of 14 review types and associated methologies. *Health information and libraries journal*, 26:91–108, 07 2009.

[GCW+24] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024.

[GF95] Michael Grüninger and Mark Fox. Methodology for the design and evaluation of ontologies. 07 1995.

[GPSF09] Asuncion Gomez-Perez and Mari Carmen Suárez-Figueroa. Neon methodology for building ontology networks: a scenario-based methodology. 01 2009.

[Hev07] Alan R. Hevner. A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19, January 2007.

[HL04] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19:281 – 316, 12 2004.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *Management Information Systems Quarterly*, 28:75–105, March 2004.

[HT19] Frank van Harmelen and Annette ten Teije. A boxology of design patterns forhybrid learningand reasoning systems. *Journal of Web Engineering*, 18(1):97–124, 2019.

[HZC+24] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024.

[KHS+17] Evgeny Kharlamov, Dag Hovland, Martin Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martín Rezk, Dmitriy Zheleznyakov, Martin Giese, Hallstein Lie, Yannis Ioannidis, Yannis Kotidis, and Manolis Koubarakis. Ontology based data access in statoil. *Web Semantics: Science, Services and Agents on the World Wide Web*, 44:3–36, 05 2017.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[Lan25] LangChain Inc. LangGraph documentation. https://langchain-ai.github.io/langgraph/, 2025. Accessed: 2026-04-15.

[LFM+26] Lauri Lovén, Reza Farahani, Ilir Murturi, Stephan Sigg, and Schahram Dustdar. Agentic edge intelligence: A research agenda. In *Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC '25, New York, NY, USA, 2026. Association for Computing Machinery.

[LWZ+24] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1, 10 2024.

[Mic25] Microsoft Research. AutoGen documentation. https://microsoft.github.io/autogen/, 2025. Accessed: 2026-04-15.

[MRV+25] Erik Miehling, Karthikeyan Natesan Ramamurthy, Kush R. Varshney, Matthew Riemer, Djallel Bouneffouf, John T. Richards, Amit Dhurandhar, Elizabeth M. Daly, Michael Hind, Prasanna Sattigeri, Dennis Wei, Ambrish Rawat, Jasmina Gajcin, and Werner Geyer. Agentic ai needs a systems theory, 2025.

[NM01] Natalya F. Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. *Knowledge Systems Laboratory*, 32, January 2001.

[NSSP26] Ume Nisa, Muhammad Shirazi, Mohamed Ali Saip, and Muhammad Syafiq Mohd Pozi. Agentic ai: The age of reasoning—a review. *Journal of Automation and Intelligence*, 5(1):69–89, 2026.

[PBPS25] Tatiana Petrova, Boris Bliznioukov, Aleksandr Puzikov, and Radu State. From semantic web and mas to agentic ai: A unified narrative of the web of agents, 2025.

[PCR+24] Marco AF Pimentel, Clément Christophe, Tathagata Raha, Prateek Munjal, Praveen K Kanithi, and Shadab Khan. Beyond metrics: A critical analysis of the variability in large language model evaluation frameworks, 2024.

[PVFIFLGC22] María Poveda-Villalón, Alba Fernández-Izquierdo, Mariano Fernández-López, and Raúl García-Castro. Lot: An industrial oriented ontology engineering framework. *Engineering Applications of Artificial Intelligence*, 111:104755, 2022.

[PVGPSF14] María Poveda-Villalón, Asunción Gómez-Pérez, and Mari Carmen Suárez-Figueroa. OOPS! (OntOlogy Pitfall Scanner!): An On-line Tool

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

for Ontology Evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2):7–34, 2014.

[Pyt24] Python Software Foundation. ast — Abstract Syntax Trees. `https://docs.python.org/3/library/ast.html`, 2024. Accessed: 2026-04-12.

[RDF25] RDFLib Team. RDFLib Documentation. `https://rdflib.readthedocs.io/en/stable/`, 2025. Accessed: 2026-04-13.

[Sch25] Johannes Schneider. Generative to agentic ai: Survey, conceptualization, and challenges, 2025.

[SRF18] Daniel Sanchez Rodriguez and Hector Florez. Model driven engineering approach to manage peripherals in mobile devices. In *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2018)*, pages 353–364, July 2018.

[SRK26] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges. *Information Fusion*, 126:103599, February 2026.

[SWMG+22] Piotr Sowiński, Katarzyna Wasielewska-Michniewska, Maria Ganzha, Marcin Paprzycki, and Costin Bădică. *Ontology Reuse: The Real Test of Ontological Design*. IOS Press, 2022.

[VRB26] Arunkumar V, Gangadharan G. R., and Rajkumar Buyya. Agentic artificial intelligence (ai): Architectures, taxonomies, and evaluation of large language model agents, 2026.

[WBZ+23] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 6 1995.

[Woo09] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 6 2009.

[WVV+02] Holger Wache, Thomas Vögele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, H. Neumann, and S. Ubner. Ontology-based integration of information - a survey of existing approaches. *Proceedings of the IJCAI'01 Workshop on Ontologies and Information Sharing, Seattle, Washington, USA, Aug 4-5*, 08 2002.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[WZ25] Christopher Wissuchek and Patrick Zschech. Exploring agentic artificial intelligence systems: Towards a typological framework, 2025.

[YEL+25] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. Survey on evaluation of llm-based agents, 2025.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.