Generative Agent Engineering: A Critical Analysis of Emerging Architectures through the Lens of Agent-Oriented Software Engineering (AOSE)

Bruno Esposito

November 5, 2025

Abstract

The advent of large language models (LLMs) has triggered the emergence of a new class of intelligent systems: generative agents. While promising unprecedented autonomy and flexibility, these agents introduce a fundamental engineering challenge: how to design, control, and make reliable systems whose decision-making core is an inherently stochastic and opaque component. This project proposes a critical and engineering analysis of emerging generative agent architectures, using the established principles of Agent-Oriented Software Engineering (AOSE) as an analytical lens. By examining four representative frameworks - AutoGen, CAMEL, Eclipse LMOS Arc and AgentLite - we aim to demonstrate how each adopts a different engineering strategy to impose structure and predictability on LLM behaviour. The analysis aims to reveal a functional convergence towards AOSE concepts, such as the reinvention of coordination artefacts similar to those of the CArtAgO model. However, it also aims to highlight a significant paradigmatic divergence' from formal organisational models such as Moise, analysing how social structures in generative systems are realised through alternative, often implicit and emergent approaches. It is concluded that a critical understanding of current architectures, informed by AOSE, is a prerequisite for the future synthesis of hybrid architectures that combine the structured deliberation of classical paradigms with the flexibility of LLMs.

Contents

Chapter 1

Introduction: The Dialectic of Agency in Software Engineering

The rise of Large Language Models (LLMs) has made possible the development of a new class of autonomous systems, also known as generative agents. These agents, leveraging the vast implicit knowledge and semantic generation capabilities of LLMs, demonstrate unprecedented fluency and versatility in tackling complex tasks, from code generation to social behaviour simulation [XCG+25][POC+23]. However, this rapid expansion of cognitive capabilities reveals an interesting 'paradigmatic divergence' from a software engineering perspective.
Modern frameworks for generative agents, while representing progress in individual problem-solving capabilities and facing similar engineering challenges, adopt an architectural approach that differs from the established principles of the AOSE paradigm. In particular, it should be noted that fundamental concepts such as agent, environment and organisation – for which AOSE has developed decades of formal models to guarantee robustness and maintainability – are now addressed with alternative and legitimate implementation approaches, often implicit and aligned with modern software development paradigms - such as message passing or the actor model -, following a path distinct from the specific experience gained at AOSE.
The literature on LLM-based agents, while placing strong emphasis on emerging capabilities such as reasoning and tool use [MDL+23][Bar24], is simultaneously proposing architectural frameworks to govern these capabilities, although with an approach that appears less formalised than the AOSE tradition [XCG+25][WMF+24][SYNG23]. Conversely, AOSE literature has historically emphasised structures and guarantees - robust coordination, explicit rules, verifiable behaviour - as fundamental elements in one's approach for building reliable and scalable multi-agent systems [WOO07][WG11]. The impressive out-of-the-box effectiveness of LLMs has prompted the development community to explore solutions in which a single LLM acts as the core, coordinator and repository of social norms, by adopting design principles that differ from those established in the AOSE field. This approach, powerful in prototyping contexts, introduces specific engineering challenges related to predictability, maintainability, scalability and security when applied to specific domains, such as mission-critical ones [HTL25].
This work argues that the future of intelligent and reliable autonomous systems may lie in a rigorous synthesis of the two paradigms, arguing for the need for an approach that integrates the semantic and generative power of LLMs within architectures explicitly modelled according to the engineering principles of AOSE, combining emergent intelligence with structural robustness.

1.1 The Long Search for the Autonomous Agent: Fundamental Definitions

The idea of "agent" is a fundamental and persistent construct in the field of computer science and artificial intelligence (AI), designed to govern the intrinsic complexity of modern software systems. An agent is defined as a computational entity located within an environment, capable of acting autonomously to pursue its design objectives [XCG+25][Woo09]. This definition distinguishes an agent from a mere object or programme on the basis of a set of key capabilities. Authors such as Wooldridge and Jennings [Woo09] have defined an intelligent agent through four canonical properties [XCG+25]:

- **Autonomy**, the agent's ability to operate without direct intervention by humans or other

agents, exercising control over its internal states and actions [XCG+25]. It is a property that extends along a spectrum, ranging from systems that rigidly execute instructions to entities that deliberate on how to achieve high-level objectives, while maintaining control over their own actions and internal states;

- **Responsiveness**, the ability to perceive its own environment and respond promptly to changes occurring within it [XCG+25] - essential for operating in dynamic and unpredictable scenarios;
- **Proactivity**, the ability to exhibit goal-oriented behaviour, taking the initiative to achieve one's goals [XCG+25] and not merely reacting to environmental stimuli;
- **Sociality**, often an agent's objectives cannot be achieved in isolation; Sociality is the ability to interact with other agents (and with humans) to achieve one's goals, either through direct communication via languages and protocols, or through indirect interactions mediated by the environment, engaging in complex activities such as cooperation, coordination and negotiation [XCG+25].

The concept of **rational agent** provides the normative basis for evaluating an agent's behaviour: an agent is rational if, for every possible sequence of perceptions, it selects an action that maximises its expected performance measure, given the evidence provided by the sequence of perceptions and any prior knowledge that the agent possesses [RNI95]. This principle establishes the ideal that the design of autonomous agents should aim for.

1.2 Two Approaches to Complex Systems Design: Historical Dialectics

The history of AI is marked by a fundamental dialectic between contrasting approaches to the construction of intelligent systems. This tension has shaped decades of research from symbolic AI to machine learning (ML) and continues to influence the debate today.
On the one hand, there is the **symbolic** and **deliberative** approach, often referred to as top-down, rooted in formal logic and the explicit representation of knowledge [XCG+25]: this paradigm posits that intelligence emerges from a process of reasoning about internal models of the world. Early AI systems embodied this philosophy, such as the Shakey robot, which built a logical model of its environment and planned its actions to achieve specific goals. However, this approach has encountered significant obstacles, particularly the problem of combinatorial explosion, where the number of possible states and sequences of actions grows at such a rate that planning becomes intractable even in moderately complex scenarios.
On the other hand, a **situated** and **reactive** - or bottom-up - approach emerged, in which researchers such as Rodney Brooks criticised the dependence on centralised models of the world, arguing that intelligence is an emergent property of the direct interaction between an agent and its environment [XCG+25][Woo09]. Emblematic architectures of this approach, such as subsumption architecture, propose building agents through a series of layers of simple reactive behaviours, which operate in parallel and are activated based on direct sensory inputs [Woo09], without the need for abstract reasoning or complex planning.
This historical dialectic highlights a 'paradox of representation', a tension that continues to characterise the field. This paradox consists in the coexistence of two paradigms of agency: on the one hand, symbolic systems, 'rich in structure but poor in knowledge', which operate on explicit and verifiable but limited models; on the other hand, LLM-based systems, which have an 'implicit structure but a wealth of knowledge', possess vast knowledge of the world but in an implicit, subsymbolic form that is not governed by an explicit architecture [SYNG23]. This dichotomy is also evident when comparing, for example, classical cognitive architectures such as SOAR - which are based on explicit procedural and semantic memories but require pre-specified knowledge -, with LLMs - which possess vast implicit knowledge of the world but lack a deliberative and verifiable reasoning structure [SYNG23].
The classical deliberative agent can be described as 'rich in structure' but limited by 'explicit knowledge', since its internal architecture – like the BDI model – is formally defined and verifiable, but its effectiveness depends entirely on the knowledge base, which must be pre-specified and codified [Woo09]. In contrast, we will see how the modern generative agent is 'rich in knowledge' but has an 'implicit structure': it has a vast and nuanced understanding of the world, derived from linguistic models, but its decision-making process is enclosed in an architecture that is often

monolithic and opaque [WMF+24][SYNG23].
Therefore, the fundamental engineering challenge shifts from how to populate a knowledge base to how to impose a governable, predictable, and verifiable structure on the vast implicit knowledge of the LLM. The focus shifts from the representation of knowledge to its governance, and it is in this transition that the thesis on the 'architectural divergence' of this analysis draws its historical foundation: modern frameworks show a tendency to prioritise generative flexibility pursuing robustness through different architectural mechanisms compared to the explicit structural robustness, typical of the AOSE tradition.

1.3 Functional Convergence and Paradigmatic Divergence

AOSE and the emerging paradigm of LLM-based generative agents represent the modern incarnations and, in some ways, the most radical expression of this historical dialectic.
AOSE can be seen as the most mature expression of the deliberative and model-based approach. Over decades of research, it has developed a corpus of rigorous engineering principles, metamodels and platforms for the design and construction of complex 'artificial societies', in which the behaviour of agents and their interactions are governed by explicit and formal specifications [WOO07][WG11].
Generative agents, on the other hand, represent a revival of the interactive and situated approach [POC+23][WMF+24]. In fact, enhanced by the generative and language comprehension capabilities of LLMs, these agents exhibit complex behaviours that emerge from linguistic interaction with the environment, with other agents and with humans.
However, modern frameworks for generative agents show functional convergence towards AOSE concepts, although implemented through implicit mechanisms specific to the paradigm (e.g. message passing). This approach represents a clear paradigm shift: it arises from the application of an alternative engineering discipline, often aligned with modern web development paradigms - e.g. microservices, reactive programming etc. - that follows a different path from the AOSE tradition [WOO07].
The analysis suggests that this divergence is a consequence of optimisation aimed at maximising the developer experience (DX) and minimising time-to-prototype. Frameworks of this type, such as AutoGen or CAMEL, offer an almost instantaneous prototype development cycle: with just a few lines of code and a well-formulated prompt, it is possible to obtain a working prototype that exhibits complex and seemingly intelligent behaviour. This rapid development cycle, which prioritises exploration and experimentation, is both an incentive for prototyping and an accelerator of innovation, allowing to exploit emerging capabilities of LLMs that would be difficult to model a priori with a rigorous AOSE approach. The AOSE approach, on the other hand, typically requires a greater initial investment in modelling and design - e.g. defining ontologies, drafting formal organisational specifications[HSB07][HSB02], and designing environmental artefacts[ORV08][RPVO09]. This initial cognitive load is aimed at obtaining guarantees of robustness and verifiability ('time-to-production'), which may appear as an obstacle in contexts focused on rapid exploration.
The analysis reveals that the architectural divergence observed is the result of a conscious design choice: robustness, traditionally achieved through explicit formal specification (AOSE), is pursued here through paradigms aligned with modern development cycles (such as message passing or dependency injection), favouring prototyping speed and Developer Experience (DX). The transition to reliable production systems raises the question of how to engineer and govern the generative power of LLMs within robust architectural frameworks, suggesting that a thoughtful synthesis of the two approaches may be beneficial.
Through this approach, the aim is to demonstrate that this trade-off, which is sustainable in the prototyping phase, raises questions about long-term scalability in mission-critical domains. A considered integration of AOSE principles could offer a way forward to address the specific architectural trade-offs of current generative agent systems, paving the way for a new generation of autonomous systems that are both intelligent and engineering-wise robust.

1.4 Research Problem and Questions

The rapid development of generative agent frameworks has followed a parallel development path, largely independent of established AOSE principles. This has led to the coexistence of two distinct engineering paradigms: on the one hand, a new generation of powerful systems with new engineering challenges (e.g. predictability); on the other, a corpus of mature engineering knowledge (AOSE) which tackles similar problems with different solutions. The central research problem is

how to engineer a robust synthesis between these two worlds, overcoming the current conceptual and methodological gap, in order to obtain more robust and engineering-wise solid generative agent systems.
To address this issue, the research is structured around one primary question and four derivative research questions:

Primary Question

(Q1) *To what extent do modern frameworks for generative agents implicitly converge, diverge, or reinvent the founding principles of AOSE, and what are the engineering implications of this relationship?*

Derived Questions

(Q2) Comparison of Cognitive Models *To what extent do modern prompting mechanisms for emergent reasoning, such as Chain-of-Thought and ReAct, functionally emulate the practical reasoning cycle of the Belief-Desire-Intention (BDI) deliberative model? What structural guarantees do the two approaches offer?*

(Q3) Analysis of the 'Plan' Concept *What are the engineering implications of transforming the 'plan' from an explicit and verifiable data structure, as defined in AOSE, to an emerging 'narrative artefact', generated linguistically by LLM agents?*

(Q4) Assessment of the 'Environment' Abstraction *How does the paradigm of the environment as a passive 'tool library', typical of generative frameworks, compare with the environment as a first-class abstraction, mediator and governor of the A&A paradigm and platforms such as CArtAgO, and what are the implications of these two choices on modularity, observability and robustness of interaction protocols?*

(Q5) Examination of the Alternative Governance Model *How does the implicit and programmatic definition of organisation in generative agent systems introduces an alternative governance model (e.g. implicit or programmatic), which differs significantly from formal specification-based AOSE approaches such as the Moise model?*

Before critically analysing these new frameworks and evaluating their architectural trade-offs, it is necessary to re-examine the fundamental pillars of AOSE, such as the concepts of agent, environment and organisation, in order to obtain the analytical tools needed to assess whether and how modern generative agents are addressing some of the already known issues.

Chapter 2

The AOSE Paradigm: Principles for Building Deliberative Systems

AOSE is the paradigm that provides a systematic, principle-based approach to the development of complex systems [WG11]. Instead of focusing exclusively on the behaviour of individual agents, it offers a systematic approach that addresses complexity through a clear separation of responsibilities at three levels of abstraction: the agent's mind, the medium of interaction and the social structure. These abstractions, in addition to being engineering constructs, provide - as will be seen - a functional cognitive macro-architecture, offering structured solutions to problems of action selection, external memory, and social governance. This chapter re-examines AOSE models in terms of their definition and as analytical tools to be used in subsequent criticism, establishing their engineering rationale.
The entire subject area of the AOSE can be interpreted as an engineering response to the challenge of building robust systems based on limited and explicit knowledge of the world - the 'paradox of representation' discussed in Chapter 1. Addressing the paradox of agents that are 'rich in structure but poor in knowledge', AOSE has focused on creating architectures and abstractions (e.g. BDI, A&A, Moise) capable of governing knowledge in a secure and predictable manner, ensuring that system behaviour remains robust and verifiable even in the face of a limited and explicit knowledge base.

2.1 The Deliberative Agent: The Belief-Desire-Intention (BDI) Model

The archetype of the autonomous deliberative agent, widely studied within AOSE, is the Belief-Desire-Intention (BDI) architecture, i.e. the pre-eminent cognitive model for practical reasoning inspired by the philosophy of mind and in particular by the work of Michael Bratman [Woo09]. The BDI model does not attempt to replicate human intelligence in all its complexity, but provides a computationally tractable abstraction of human decision-making based on three fundamental mental components [Woo09]:

- **Beliefs** represent the set of information that the agent possesses about the world, about himself and about other agents. They may be incomplete or incorrect and are constantly updated through perception [Woo09];
- **Desires** represent the agent's motivational state; they are the states of the world that the agent would like to achieve, i.e. its potential goals. An agent may have multiple and potentially conflicting desires [Woo09];
- **Intentions** represent the deliberative state of the agent, they are the desires to which the agent has committed. Unlike desires, intentions actively guide the agent's behaviour [Woo09]. Their key property is persistence: an agent does not abandon an intention lightly, but persists in pursuing it until it is achieved, becomes clearly unachievable, or the reason for which it was adopted ceases to exist. This commitment allows the agent to focus its computational resources and not continually reconsider its decisions.

The behaviour of a BDI agent is governed by a practical reasoning cycle, often implemented as an interpreter that operates continuously. In this cycle, the agent: perceives events from the

environment, updates its beliefs, generates options (desires), selects a coherent set of intentions, and finally chooses an intention to pursue. The execution of an intention occurs through the activation of a plan, i.e. a predefined sequence of actions or sub-objectives. The plans, stored in a library, constitute the agent's procedural knowledge: they are predefined, reusable, and verifiable procedures, each associated with a triggering event and a context of applicability that govern their selection and execution in response to changes in the agent's beliefs or objectives [Woo09].

2.2 The Mediating Environment: The Engineering Ratio of a First-Class Abstraction

In the AOSE paradigm, the environment ceases to be a passive backdrop and becomes a first-class engineering abstraction: a programmable, stateful, and observable medium that mediates interactions between agents and their access to resources, thus ensuring their decoupling [WOO07]. The Agents & Artifacts (A&A) meta-model formalises this vision, proposing an environment populated by agents and artefacts, i.e. non-autonomous computational entities explicitly designed to mediate interactions between agents and facilitate their access to resources [WOO07]. The engineering rationale behind this approach, as implemented in platforms such as CArtAgO, offers key benefits that are widely discussed in the AOSE literature [WOO07]:

- **Modularity and Reusability**, the coordination logic is encapsulated in artefacts, not rigidly encoded within agents. An artefact that implements an auction mechanism, for example, can be reused in different agent companies without requiring any changes to the agents themselves. This separation between the agent's business logic and the coordination logic of the interaction is a pillar of robust software engineering [WOO07];
- **Observability and Verifiability**, the state of an interaction (e.g. current bids in an auction, the order of speaking in a meeting) is an observable property of the artefact. This enables runtime inspection, debugging, and formal verification of interaction protocols [WOO07]. Without an explicit entity that reifies the state of interaction, collective behaviour becomes a black box, difficult to analyse and certify;
- **Guided Interaction**, artifacts enforce interaction rules (e.g. preventing an agent from speaking out of turn) through programmable usage policies. This makes collective behaviour more predictable and robust, since the 'laws of social physics' are enforced by the environment itself [WOO07] and do not depend on the goodwill or correct implementation of each individual agent.

In summary, treating the environment as a first-class entity populated by programmable artefacts transforms the coordination problem from a complex negotiation between agents to a problem of designing robust and reusable environmental mechanisms.

2.3 The Explicit Organisation: The Moise Model as an Analytical Tool

AOSE extends its engineering scope beyond the individual agent and interaction environment to explicitly model the social structure of the multi-agent system. Instead of allowing hierarchies and group dynamics to emerge implicitly, models such as Moise provide a formal language for specifying the organisation of multi-agent systems [HSB07]. This approach allows the social structures of the system to be designed, verified and dynamically reconfigured. The Moise model is based on three orthogonal dimensions [HSB07][HSB02], which clearly separate the different organisational competences:

1. **Structural Specification**, defines who is part of the organisation and how its members are connected. This specification introduces roles as abstractions of expected behaviours (e.g. 'reviewer', 'author'), groups as aggregations of collaborating roles, and social ties as relationships of authority (who can give orders to whom) and communication (who can talk to whom) [HSB07][HSB02];
2. **Functional Specification**, defines what the organisation does. It starts from the overall objectives of the system and breaks them down hierarchically into social schemes. These diagrams represent high-level plans that are further broken down into missions, i.e. specific

objectives that must be achieved. The missions are therefore assigned to the roles defined in the structural specification [HSB07][HSB02];

3. **Deontic Specification**, defines the rules of the game, linking responsibilities (roles) to objectives (missions). This specification uses deontic concepts to define the rules governing the behaviour of agents. Specify which missions are mandatory or optional for an agent who adopts a specific role [HSB07][HSB02].

The engineering rationale behind an explicit, machine-readable organisational specification, such as Moise's, offers crucial advantages [HSB07][HSB02] that serve as benchmarks for the critical analysis of modern systems:

- **Runtime Reasoning**, agents can consult the organisational model to understand their roles, responsibilities and with whom they are authorised to interact. An agent who is aware of the organisation can adapt his behaviour according to his position and social duties [HSB07][HSB02];
- **Dynamic Reorganisation**, an explicit model allows the organisation to change at runtime (e.g. by adding new roles, modifying lines of authority) without shutting down and redeploying the entire system [HSB07][HSB02]. This agility is essential for systems that need to operate in open and dynamic environments;
- **Deontic verification**, the separation of structural, functional and deontic specifications allows for formal verification of the company's behaviour. For example, it is possible to verify that an agent cannot perform an action for which it does not have permission in its current role [HSB07][HSB02], ensuring that agents, by adhering to the rules, will contribute to the achievement of overall objectives.

These engineering principles of modularity, observability, verifiability, and explicit governance outlined above constitute the analytical benchmark against which Chapter 4 will examine the architectures and mechanisms of the emerging paradigm of generative agents to understand whether and how the solutions adopted by modern frameworks compare with these fundamental properties, considered central by AOSE for the construction of robust and reliable systems.

2.4 Mapping Cognitive Models: The AOSE Structure as Architecture of the Mind

The engineering principles of AOSE are not just software constructs, but can find a parallel in computational models of human cognition. The tripartite Agent-Environment-Organisation architecture can be interpreted as a cognitive macro-architecture that provides an explicit structure to components that are often merged and implicit in modern linguistic agents. In this section, we propose an interpretation of AOSE architecture as a cognitive macro-architecture, to highlight how it provides engineered solutions to fundamental problems in artificial cognition. This perspective is supported by the analysis of Sumers et al. (CoALA), who use cognitive architectures as a lens to analyse and structure the emerging field of linguistic agents, highlighting the need for distinct memory modules, action spaces and decision-making procedures [SYNG23]. Similarly, AOSE's emphasis on the environment as a mediator of social interactions finds a parallel in the models of situated and social cognition discussed in cognitive science.

The BDI Cycle as an Action Selection Mechanism

The practical reasoning cycle of the BDI model is a computationally tractable implementation of the action selection mechanism (decision-making procedure), a central component of every cognitive architecture [Woo09]. It formalises the process through which an agent, given their representation of the world (Beliefs) and their objectives (Desires), commits to a course of action (Intentions) and pursues it. This contrasts with the often opaque and monolithic decision-making process of purely generative agents.

The A&A Environment as Structured External Memory

The AOSE environment, populated by artefacts (CArtAgO), is not only a space for action, but can also serve as a structured external memory that supports and lightens the agent's working

memory. An artefact, by maintaining the state of an interaction (e.g. an auction, a shared document), reifies information that the agent would otherwise have to manage entirely in its internal state. This outsourcing of the state, governed by formal interfaces, ensures greater reliability than prompt-based context management, which is inherently volatile and limited.

Moise as Social Knowledge and Explicit Norms

A Moise organisational specification can be seen as an explicit, machine-readable representation of social and normative knowledge which, in a cognitive architecture, would reside in long-term semantic memory. Instead of having to infer social rules from the vast and unstructured pre-training corpus, a Moise-aware agent can directly query its organisational structure to understand its roles, permissions, and obligations [HSB07][HSB02]. This provides a mechanism for governing social behaviour that is verifiable and robust, offering an alternative to the implicit (or 'emergent') governance model of systems in which rules are defined only through prompts.

Chapter 3

The Emergence of Generative Agents: A New Paradigm Driven by Large Language Models

The rise of LLMs is redefining the design of intelligent agents [XCG+25][WMF+24] by leveraging their emerging capabilities to equip systems with new faculties of understanding, reasoning, and interaction. This chapter outlines the architecture of these new agents and analyses the transformation undergone by the concept of deliberation, moving from a structured process to an emerging linguistic activity.
If AOSE responds to the 'structure-rich' side of the paradox, the generative agent paradigm mirrors the 'knowledge-rich' side. In this paradigm, the engineering challenge shifts to attempting to impose a governable structure on the implicit knowledge of the LLM. This chapter, in addition to analysing the architecture and deliberative mechanisms of these new agents, will show how the solutions adopted to manage the challenges of this paradigm (e.g. stochastic nature) lead to mechanisms that converge functionally with concepts already formalised by AOSE, highlighting the functional convergence and, at the same time, a paradigmatic divergence in their implementation.

3.1 The Generative Agent: Deliberation as a Linguistic Process

A generative agent is not simply an LLM; rather, the LLM serves as a central component - the 'brain' or computational core - within a broader architecture designed to overcome the inherent limitations of a stateless language model [XCG+25][SYNG23]. Some of the most recent surveys, including those by Wang et al. and Xi et al., converge on a modular architecture comprising three key elements [XCG+25][WMF+24][MDL+23][GCW+24]:

LLM as a Computational Core

The LLM is the engine of reasoning because its vast knowledge of the world, pre-trained on enormous text datasets, and its ability to generate coherent and contextually appropriate language, provide the foundation for all of the agent's other capabilities [XCG+25][WMF+24].

Hybrid Memory System

To overcome the fundamental limitation of an LLM's finite context window, generative agents implement a multi-level memory system inspired by models of human cognition [POC+23][SYNG23].

- **Working Memory** corresponds to the immediate context provided to the LLM via the prompt [SYNG23][Wen23]. It may contain current perceptions, active goals, and information retrieved from long-term memory;
- **Long-Term Memory** is an external mechanism for the persistence of information. It can be divided into episodic memory (the chronology of past experiences), semantic memory (acquired factual knowledge) and procedural memory (skills implicit in LLM weights and

explicit in agent code) [POC+23][SYNG23]. Typically, it can be implemented using vector databases that store information in the form of embeddings, allowing efficient retrieval based, for example, on relevance [WMF+24][Wen23].

Action Space and Tool Use

One of the most significant innovations is the ability of generative agents to 'act' on the world [XCG+25][WMF+24]. This goes beyond simple text generation, as the agent can be equipped with a set of tools — which may be external APIs, code interpreters, search engines, or other specialised models — that it can decide to invoke [MDL+23][MBSC24]. This process is known as grounding and anchors the agent's linguistic capabilities to the external world [XCG+25][SYNG23], allowing it to access up-to-date information, perform calculations or carry out concrete actions in digital or physical environments.

3.1.1 From Memory to Reflection: The Genesis of Long-Term Plans

A memory system based solely on the retrieval of raw experiences is not sufficient to generate consistent behaviour oriented towards long-term goals. To overcome this limitation, the most advanced cognitive architectures for generative agents introduce a reflection mechanism [POC+23]. This process, inspired by human cognition, is a periodic cycle in which the agent does not merely perceive and react, but synthesises retrieved memories to generate higher-level inferences. Operationally, the agent is prompted to ask themselves salient questions about their past experiences (e.g. 'What are my main aspirations?' or 'What have I learned from my recent interactions?') [POC+23]. The answers to these questions, which are summaries and abstractions of the raw observations, are in turn stored as new higher-level memories [POC+23]. This process enriches the knowledge base and creates a hierarchy of inferences that allows the agent to construct a self-representation and more robust long-term plans [POC+23]. This mechanism can be seen as the functional analogue, albeit operationally very different, of the deliberation process in the BDI model, where an agent reconsiders their beliefs to form new intentions.

3.1.2 From Planning to Prompting: The Linguistic Shift in Agent Deliberation

The fundamental divergence between the AOSE paradigm and that of generative agents lies in the conception and implementation of the deliberative process. In AOSE, planning is an explicit and structured process based on the execution of a data structure (a plan). In generative agents, planning becomes more of an emergent linguistic process, dynamically generated by the LLM.

Chain-of-Thought (CoT) Prompting

This technique represents a significant first step which, instead of asking an LLM only for the final result, prompts it to generate a 'chain of thought', i.e. a sequence of intermediate reasoning steps leading to the solution [WWS+22]. In this way, breaking down a complex problem is no longer an algorithmic operation on a data structure, but an act of linguistic generation. The generative agent produces a sequence of intermediate reasoning steps, and the generated text serves both as an observable trace of the deliberative process and as context for subsequent steps. The plan is no longer an explicit data structure to be executed, but an emergent linguistic process that guides behaviour through the sequential and autoregressive generation of reasoning steps [WWS+22].

ReAct (Reason + Act)

The ReAct pattern further refines this approach by creating synergy between reasoning and action [YZY+22]. The deliberative process is structured as an iterative cycle of Thought-Action-Observation:

- **Thought**, the agent generates internal reasoning to analyse the situation, evaluate progress and decide on the next move. It represents an action in the realm of language (reasoning) that does not influence the external environment but serves to compose useful information by reasoning about the current context to support future actions [YZY+22];

- **Action**, the agent translates the thought into a concrete action, typically the invocation of a tool (e.g. Search['question']). It represents the way in which the agent interacts with external sources/environments, guided by reasoning traces ('reason to act') [YZY+22];
- **Observation**, the agent receives the output of the tool (e.g. the search result) and integrates it into its context [YZY+22]: is what the agent receives from the environment after an action and which becomes part of the context [YZY+22].

This cycle repeats itself, with each observation feeding into the next thought. Therefore, planning is no longer an internal monologue (as in CoT), but a dynamic dialogue with the environment, and the agent continuously adapts its strategy based on the feedback it receives, making its behaviour much more robust and flexible than a static plan [YZY+22].

3.1.3 Limits and Challenges of Emerging Reasoning

While emerging reasoning mechanisms such as CoT and ReAct represent a qualitative leap in the deliberative capacity of linguistic agents, an engineering analysis highlights their implementation challenges (e.g. susceptibility to propagation error), which is widely documented in scientific literature. These approaches, although overcoming the limitations of direct prompts, introduce new challenges related to their linguistic and unstructured nature, which directly impact the robustness and reliability of the system.

The Challenges of Chain-of-Thought (CoT)

By inducing the model to 'think aloud', CoT breaks down a complex problem into intermediate steps, making the reasoning more explicit. However, this strategy presents a fundamental risk: there is no guarantee that the reasoning paths are correct, as they can lead to both correct and incorrect answers [WWS+22]. Research has shown that this approach is prone to 'hallucinations', a phenomenon that leads to the generation of information that is 'factually incorrect or illogical' or 'incorrect, meaningless or unreal' [Bar24].
This phenomenon, known as error propagation, is particularly critical: a single misstep at the beginning of the chain of reasoning can 'contaminate' the entire deliberative process [YZY+22], leading to an incorrect conclusion even if the subsequent steps are logically consistent with the incorrect premise. As highlighted in the comparative study on ReAct, hallucination represents the main mode of failure for CoT, as the latter is not anchored to external feedback [YZY+22]. This issue impacts the agent's reliability, since the validity of their action plan depends entirely on the correctness of a purely internal and unverified chain of inferences.

The Limitations of ReAct's Anchorage

The ReAct paradigm attempts to mitigate CoT hallucinations by anchoring reasoning to concrete observations from the external environment [YZY+22], through the invocation of tools. The Thought-Action-Observation cycle creates a feedback mechanism that, in theory, should keep reasoning 'grounded' and improve reliability. However, even this approach, while representing progress, introduces new and significant challenges:

- **Derailment from Misleading Observations**, the effectiveness of ReAct is directly proportional to the quality of the information obtained from the environment. One of the most common errors identified in its assessment is the 'search result error', where an uninformative, ambiguous, or misleading observation can derail the entire reasoning process [YZY+22]. The agent, not having a structured model of the world to assess the plausibility of the observation, accepts it as fact, formulating the subsequent thought on an erroneous basis;
- **Loops and Inefficient Reasoning**, although ReAct is more structured than CoT, its flexibility can lead to inefficiencies. It has been observed that the model can become stuck in an execution loop, repeatedly generating the same thoughts and actions without being able to break out of a stalemate and complete the task [YZY+22]. This inefficient planning is one of the ways in which the system fails;
- **Structural Constraints and Reduced Flexibility**, the close interleaving between thought, action and observation, while improving concreteness, reduces the flexibility of reasoning compared to pure CoT. This structural constraint can lead to a higher rate of 'reasoning errors'

[YZY+22], where the agent fails to formulate complex logical steps because it is forced to rigidly follow the Thought-Action-Observation pattern.

While these mechanisms extend the capabilities of agents, they do not eliminate the need for architectural governance. The stochastic nature of these linguistic processes - their susceptibility to hallucinations, error propagation, and derailment from imperfect inputs - reinforces the argument that the vast implicit knowledge of LLMs needs to be harnessed within engineering frameworks that guarantee robustness, verifiability and predictability, a goal for which AOSE has been developing formal solutions for decades. In fact, this need is also recognised outside the AOSE field: for example, recent surveys on LLM-based agents propose unified frameworks with explicit modules for planning, memory and action [XCG+25][WMF+24], and works such as COALA [SYNG23] use classical cognitive architectures to structure and give coherence to agent behaviour, overcoming the limitations of a 'raw' LLM.

3.2 The Environment as a 'Tool Library': Challenges in Action and Memory

The current implementation of "Tool Use" presents specific engineering challenges in generative agent systems, as most frameworks treat a tool as a simple function (e.g. Python) associated with a natural language description (docstring) [MBSC24] and the agent, relying solely on this description, must perform two steps with a potentially high risk of failure: **call generation**, whereby the LLM must generate a syntactically correct (e.g. a JSON object) and semantically appropriate call, a process known for its sensitivity to minor variations in the prompt (prompt brittleness); **output parsing**, whereby the agent must interpret output that is often unstructured text, with no formal guarantee as to its format or content.
This interaction, mediated exclusively by natural language, introduces a trade-off: it offers great semantic flexibility but shifts the burden of reliability onto the interpretation of the LLM. It represents a paradigmatic divergence from the AOSE A&A approach, which addresses the same engineering challenge (interaction with resources) by elevating the interface to a first-class abstraction (the artefact [ORV08]), which favours formal contracts and programmatic interfaces. An artefact is not just a simple "tool", but a computational entity with:

- **A Formal Interface**, a defined set of operations, observable properties, and events that constitute an explicit contract between the agent and the resource [ORV08][RPVO09][ORV+04];
- **Observable Internal State**, the state of the interaction (e.g. a file is open, an auction is in progress) is a property of the artefact, which can be inspected at runtime [RPVO09][ORV+04];
- **Programmable Usage Policy**, the artefact itself imposes the rules of interaction, preventing improper use (e.g. writing to a file that is not open) [RPVO09].

So, while a generative agent 'hopes' to use a tool correctly based on semantic interpretation, an AOSE agent can interact with an artefact through a robust and verifiable protocol.
This same challenge, rooted in interaction based on semantics rather than formal contracts, extends beyond action and is also evident in the way modern agents manage memory. The retrieval-based approach to memory - such as Retrieval-Augmented Generation (RAG) - introduces a fundamental difference compared to the AOSE approach. It may seem that modern generative agents achieve a similar result through hybrid memory mechanisms - such as RAG [XCG+25][POC+23] - which overcome the volatility of the prompt context alone. However, there is one fundamental difference: retrieval in a RAG system is typically driven by semantic relevance criteria, where the agent retrieves memory chunks that are vectorially similar to the current query. Although powerful, this mechanism offers no formal guarantees regarding interaction [POC+23][WMF+24] since the selection and use of retrieved information are left to the interpretation of the LLM, which is subject to failures or hallucinations.
On the contrary, a CArtAgO artefact is a reactive and governed computational entity: it not only stores a state, but also encapsulates the operational logic and interaction rules, exposing formal operations with defined pre-conditions and post-conditions [RPVO09]. Interaction is not based on semantic similarity, but on the invocation of operations whose effect on the environment is engineered and predictable. The artefact acts as a software contract that governs interaction offering structural robustness through a formal interface, an engineering approach that differs from that based on semantic retrieval.
In both cases, whether in action (tool) or state management (memory), a common pattern emerges

that demonstrates the underlying paradigmatic divergence: the environment is conceptualised differently from the first-class mediating abstraction [WOO07], taking the form of a set of functions or APIs that agents can invoke directly [MBSC24] according to an agent-centric approach. Coordination, instead of being entrusted to explicit environmental artefacts (as in CArtAgO), is managed by the internal logic of the agents themselves or, as we shall see, delegated to specialised agents with the role of 'manager' or 'coordinator' [MBSC24]. This design choice closely couples agents, implementing alternative mechanisms for the governance, modularity and reusability - often based on message passing or dependency injection - which differ from those that AOSE has identified as fundamental for building reliable systems [WOO07]. This structural feature will be further highlighted in the analysis of the source code in Chapters 5 and 6.

3.3 The Implicit Organisation: An Alternative Governance Model

Emerging frameworks for building generative agents [WBZ+24][LHI+23][Fou] provide concrete implementations of this new paradigm. A critical analysis of these systems, in light of the engineering principles outlined in Chapter 2, reveals a common trend: the concepts of environment and organisation are implemented implicitly and programmatically.
Team structures, hierarchies, and interaction protocols do not derive from a formal and verifiable organisational specification such as Moise [HSB07][HSB02]. Instead, they are defined through code configurations and, above all, through natural language prompts that instruct agents on their roles, responsibilities and modes of interaction. For example, in many of these systems, a specialised role such as 'coordinator' or 'planner' is defined via a prompt, and the logic governing turn-taking or task assignment is encapsulated within a specific 'manager' agent, rather than in an independent environmental artefact.
This approach, which prioritises flexibility in programmatic configuration and natural language and differs from the rigour of formal specifications, defines an alternative governance model not based on explicit declarative specifications. By choosing not to adopt a formal organisational dimension [HSB07], coordination logic, structural reasoning and the application of social rules are implemented through alternative engineering mechanisms. This solution, aligned with modern software development paradigms, shifts governance responsibility from the formal specification level (AOSE) to the programmatic implementation level (modern frameworks). This highlights a clear paradigmatic divergence and defines a different set of engineering challenges, as discussed in recent surveys [GCW+24][HTL25].
The following table summarises the key architectural differences discussed.

Table 3.1: Summary of key architectural differences

Architectural Dimension	AOSE Paradigm	Generative Agents Paradigm
Agent Model	Deliberative, based on explicit mental states (beliefs, desires, intentions) and practical reasoning [Woo09]	Cognitive-Linguistic, with the LLM as the core of reasoning and a hybrid memory (working, episodic, semantic) [POC+23][WMF+24][SYNG23]
Planning Process	Explicit, based on the execution of a data structure (plan) from a predefined library [Woo09]	Emergent, based on the generation of a linguistic process (e.g. CoT, ReAct) [WWS+22][YZY+22]
Concept of Environment	First-class abstraction, active, mediating and governed (Artefacts). Explicit coordination locus [WOO07]	Set of tools and APIs, often passive, invoked by the agent. Coordination implicit in agent logic [MDL+23][MBSC24]
Concept of Organisation	Explicit, formal and verifiable specification (Structural, Functional, Deontic). The organisation is a first-class entity [HSB07][HSB02]	Implicit, emerging from interactions or defined by prompts and code configurations. Governance managed programmatically (e.g. through manager agents, dependency injection) rather than through formal specifications."

The analysis conducted so far has highlighted a picture in which, on the one hand, there are emerging cognitive capabilities and unprecedented flexibility; on the other, governance challenges

and architectural trade-offs that result from design choices that favour an agent-centric or service-oriented paradigm distinct from that consolidated in AOSE. To move from this critical assessment to concrete evidence, Chapter 4 will provide a detailed overview of the architectural characteristics of the frameworks analysed, while Chapters 5 and 6 will analyse the relevant source code in detail. Specific implementations of coordination and organisation mechanisms will be examined to demonstrate how the AOSE logic is implicitly implemented using different architectural paradigms, showing a clear paradigmatic divergence from AOSE.

Chapter 4

Critical Analysis of Frameworks and Architectures of Generative Agents

The advent and rapid proliferation of large language models (LLMs) have ushered in a new era in the development of intelligent systems, giving rise to the concept of generative agents. These agents, empowered by the capabilities of LLMs, seem able to perform complex tasks, interact with the environment and collaborate in ways that were previously difficult to achieve - e.g. through natural language. The increasing sophistication of such agents has made evident the need for dedicated tools and software architectures - frameworks - that facilitate their design, development, orchestration and management. Recent literature bears witness to fervent research and development in this area with the proposal of numerous frameworks, each with distinct approaches and philosophies reflected in the agent architectures they promote or enable.

This chapter conducts an in-depth engineering analysis and critical comparison of some of the main currently available generative agent frameworks - such as AutoGen, CAMEL, Eclipse LMOS Arc and AgentLite - with the aim of going beyond the description of their functionalities and aiming to explore the 'why' behind certain architectural choices, the potential inherent trade-offs and their implications on the properties of agent systems. The analysis will focus in particular on the agent models and architectures they enable, the multi-agent management capabilities and how the distinctive parts of each framework have been designed to include and address the inherent characteristics of LLMs. Understanding these aspects is essential to comprehend how each framework attempts to impose structure and predictability on the behaviour of LLMs - balancing emerging capabilities with the need for reliable and controllable system operation - by proposing different (albeit similar) approaches to the definition of agents.

The following analysis does not have a purely descriptive or theoretical value, the engineering criteria defined here and used for the evaluation of frameworks - such as modularity, transparency, ease of debugging and maintainability - will form the methodological basis for the next phase of this work. As detailed in Chapter 5, these same criteria will be operationalised into structured qualitative evaluation metrics used for the empirical and engineering comparison between the BDI approach and generative agent-based implementations.

4.1 Analysis Methodology and Criteria for Framework Selection

Given the innovative nature and rapid evolution of the field, the analysis conducted followed an iterative and pragmatic methodological approach. The research started with a reconnaissance phase and in-depth study of the scientific literature - including theoretical and practical papers - to consolidate the conceptual foundations and identify the most relevant frameworks in the LLM-based agent landscape. Subsequently, the most relevant frameworks have been selected, favouring those that offered practical aspects and allowed for a more holistic implementation of this type of agents. From this list, some of the most important and relevant frameworks in the literature were initially chosen, such as AgentLite, AutoGen and CAMEL.

AgentLite was used as a 'pilot framework' to develop and refine the analysis methodology. Its relative simplicity and comprehensiveness with respect to the main aspects of a generative

agent, coupled with the availability of implementation examples, made it possible to explore and understand the underlying mechanisms and test a systematic approach. This experience guided the definition of a structured methodology, which was then applied to all the frameworks examined:

1. **Analysis of documentation and official literature**: in-depth study of research papers and official documentation to understand the vision and agent architecture promoted by each framework;

2. **Analysis of code and implementation examples**: examination of the main source code relating to the key concepts and architectural components of the agents, together with analysis of the examples provided, in order to obtain practical feedback on the theoretical and documentary statements;

3. **Creation of diagrams**: development of component, class and sequence diagrams to visualise and better understand the structure, architecture and main operational flows of each framework and its agents;

4. **Creating and updating tables**: maintaining and continuously updating summary tables to track general and most important characteristics and extract potential strengths and weaknesses.

This methodology made it possible to navigate the complexity of the field, trying to ensure a thorough and evidence-based analysis. The decision to consider AutoGen, CAMEL and Eclipse LMOS Arc as the main frameworks to be analysed was guided by a number of criteria aimed at ensuring a representative coverage of the state of the art. Firstly, the relevance and impact that these frameworks have demonstrated or promise to have in the research community and/or industry was considered. Secondly, the availability of detailed technical documentation and access to the source code were considered crucial to enable an in-depth analysis of the underlying agent architectures. A third crucial criterion was the ability of the selected frameworks to represent distinct approaches and architectures for building agents: AutoGen focuses on 'conversation programming' to define the conversational agent; CAMEL emphasises the 'role-playing' and 'inception prompting' to shape the internal architecture and behaviour of the agent; finally, Eclipse LMOS Arc proposes an 'operating system for LLM' that provides the environment for agents defined via a specific DSL. This inherent diversity in the selected paradigms addresses the need to sample different engineering approaches to the construction of generative agents and their internal architectures. It allowed to extract common patterns in agent architectures, identify fundamental divergences and understand engineering challenges from multiple perspectives, thus offering a broader view. Finally, the potential of each framework was assessed in illustrating different engineering challenges and solutions specific to the development of generative agent architectures.

In addition to the main frameworks, the chapter includes a brief analysis of AgentLite which was selected not only as a 'pilot' project, but also because it offers a complementary perspective, characterised by a lightweight and strongly research-oriented approach, which results in a minimalist yet explicit agent architecture in its core components.

4.1.1 Critical Source Evaluation and Temporal Context

The speed with which the field of generative agents and LLMs is developing often means that fundamental works are initially available as preprints (e.g. arXiv) or directly as project documentation on platforms (e.g. GitHub) and their official websites. The case of the Eclipse LMOS Arc is emblematic in this respect since its primary information comes mainly from its project documentation on GitHub and that on its official website, rather than from peer-reviewed academic publications. Indeed, although general surveys on agent protocols mention LMOS, these do not delve into the architectural details of the framework itself.

Thus, the further inclusion of frameworks whose documentation is mainly project-based was considered necessary to provide an up-to-date view of the state of the art as it allows for the analysis of the latest and most relevant technologies, while maintaining a critical discussion on the credibility and maturity of the sources.

It should also be noted that information on the AutoGen framework is up to date with version 0.5.6 of its documentation and official code in Python. For the other frameworks (CAMEL, Eclipse LMOS Arc and AgentLite), the analysis is based on the information available in the documentation and code provided up to the end of August 2025.

4.2 AutoGen: The Conversational Agent

4.2.1 Vision and Architecture of the Framework

AutoGen, developed by Microsoft, is an open-source framework designed to simplify the creation of LLM-based applications through multi-agent conversations. The main objective is to facilitate the development of complex agents and workflows that integrate LLM, human input and external tools, exploiting the ability of advanced language models to process information, receive feedback and progress dynamically within a conversational flow.

The core principle of AutoGen is 'conversation programming', the paradigm that defines the AutoGen agent as a conversational entity with a specific role and capabilities, whose behaviour and interactions are programmed through the management and control of the conversation itself. Such agents represent configurable software entities capable of sending messages, reacting to received messages and responding using LLM, human input or the output of external tools, which become an integral part of the agent's capabilities.

The architecture of the AutoGen framework is structured on two main levels that offer different degrees of abstraction and control for the realisation of conversable agents:

1. **autogen-core** represents the underlying engine of the framework, based on an event-driven architecture and the actor model. It consists of the lowest level of the AutoGen architecture that provides the communication infrastructure - AgentRuntime - and manages the agent lifecycle;

2. **autogen-agentchat** offers a higher-level API specifically designed for building conversation-based multi-agent applications, providing abstractions for conversational agents and interaction patterns.

This dual architecture balances the power and customisation offered by Core with the ease of use of the AgentChat API to define and orchestrate agents.

4.2.2 Agent Architecture

The architecture of an agent in AutoGen reflects the division between autogen-core and autogen-agentchat, allowing agents with different levels of abstraction and control to be defined, balancing the flexibility of the underlying engine with the ease of use of the high-level API.

The autogen-core Level Agent

At the autogen-core level, the agent is designed as an autonomous and independent software entity based on the actor model. This design, which models each agent as an actor with its own state that processes messages asynchronously, is suitable for building complex, distributed workflows. Moreover, the framework provides the basic communication infrastructure - AgentRuntime -, while the agents are primarily responsible for their own internal logic.

Definition and Role
The fundamental component at this level is the base class RoutedAgent, which acts as a router capable of receiving messages and routing them to the correct function (or 'handler') according to the type of message received. This logic is evident in the on_message_impl method, which determines the type of the incoming message and looks for a corresponding registered handler.

In order to define a new type of agent, a developer creates a subclass of RoutedAgent and implements specific methods to handle the different types of messages the agent expects to receive.

Key Components
Base class (RoutedAgent)
Agents in the Core can inherit from RoutedAgent, which provides the central infrastructure for message management. During initialisation, the agent automatically discovers and registers all methods designated as message handlers via the _discover_handlers method.

Message Handlers
The specific logic of an agent is defined within methods decorated with @event or @rpc, which register the method as a callback function for a specific type of message. The runtime, via the RoutedAgent, will know which of these functions to send a specific message to: for example, a

method defined as async def handle_event_message(self, message: Message, ...) will only be invoked when the agent receives an object of type Message.

Conditional Routing
The @rpc decorator may include a 'match' parameter capable of accepting a function that allows even more specific routing: the handler will only be executed if the message type matches and the 'match' function returns True. This allows different behaviour to be defined for messages of the same type but with different contents.

Agent Identity
Each agent instance is identified by a unique agentid, consisting of an agent type and a key. The agent type corresponds to the registered agent class, while the key allows multiple instances of the same type (e.g. separate sessions) to be distinguished.

Flexible API and Composition
The Core API is 'unopinionated', which means that developers can integrate agents of any kind: for instance, it is possible to integrate autogen-agents via a wrapper or to implement fully customised agents without modifying the underlying runtime.

Operating Cycle
Asynchronous Message Management
Message handler methods are asynchronous functions (async def) that allow multiple requests to be handled concurrently: an asynchronous handler processes each message and can call other agents or services without blocking the runtime.

Runtime Registration and Life Cycle
Agent types must be registered with the runtime before execution via the register() method, which associates an AgentType (a unique string) with a factory function that creates instances of that agent. This way the runtime knows how to instantiate agents on the fly and controls the agent life cycle: when a message arrives addressed to an agent, the runtime retrieves the corresponding agentid instance or creates a new one if it does not already exist. This 'on-demand' approach facilitates scalability and dynamic management of agents.
In summary, the autogen-core layer provides the low-level foundation for communication, state management and the agent lifecycle, enabling the creation of distributed, scalable and resilient multi-agent systems.

The autogen-agentchat Level Agent

The autogen-agentchat module builds on the capabilities of the core layer to offer a higher-level API for implementing agents and multi-agent applications. In addition, it provides predefined options for agents, through preset behaviours, and for groups of agents - called Teams - through predefined multi-agent design templates.

Definition and Role
The key agent in this layer is the ChatAgent, which embodies the entity capable of participating in conversations. All agents in autogen-agentchat derive from common base classes - such as the ChatAgent -, possess attributes such as name and description, and their role can be defined via a system_message in which the behaviour and specialisation in the conversation can be specified via prompts.

Key Components
Stateful Agent
Agents maintain an internal state that is updated with each call to the run() method with new messages from the history, allowing them to 'remember' the conversation they have had up to that moment. The state can be serialised and deserialised using the save_state() and load_state() methods for persistence.

LLM Integration
The LLM is the main engine for message understanding, reasoning and response generation. The agent manages the conversational context to be sent to the LLM using configurable context windows, such as BufferedChatCompletionContext (for the last N messages) and TokenLimitedChat-

CompletionContext (for token-based limits).

Use of Tools (Function Calling)
Agents can call external functions (tools) to perform specific operations. These tools are defined as FunctionTools that the LLM can choose to invoke and can be of various types: Python functions, PythonCodeExecutionTool to execute Python code in Docker containers, search tools such as GraphRAG, HttpTool for HTTP calls, etc. This mechanism allows agents to access external information or perform specific actions, directly addressing the limitations of LLMs in interacting with the real world and managing external knowledge. AutoGen also supports the Model Context Protocol (MCP) via McpWorkbench for structured tool management.

Human Input (Human-in-the-loop)
The UserProxyAgent acts as an interface for human input by optionally allowing a user to actively participate in the conversation by providing feedback or manual responses during execution. This agent pauses the flow until the user (or an external UI) responds, allowing real-time interactions and integrating human control as an active component in the agent's conversational architecture.

Memory
The primary form of memory for an AutoGen agent is the conversation history itself. In addition to this context, the framework also supports a memory protocol for storing and retrieving relevant information, facilitating Retrieval-Augmented Generation (RAG) patterns where relevant information is retrieved from external databases and added to the agent's context at the appropriate time.

Structured Output
Allows LLM models to return structured JSON text with a predefined schema provided by the application. In contrast to JSON mode, the schema can be supplied as a Pydantic BaseModel class, which can also be used to validate the output. This type of output may also be useful for incorporating Chain-of-Thought into agent responses.

Operating Cycle
The behaviour of an autogen-agentchat agent is inherently guided by a structured conversational flow, with the aim of processing an incoming message and generating a relevant response. Taking AutoGen's main ChatAgent implementation - AssistantAgent - as a reference, the process consists of several sequential steps:

1. **Receiving and Contextualising Messages**: when receiving new messages, the first step is to integrate them into its internal context, which acts as a short-term memory of the conversation;
2. **Enriching Context with Memory**: next, the agent draws on its long-term memory and, if configured, memory modules are queried to retrieve relevant information that can enrich the context before it is sent to the LLM model. This step is crucial for implementing Retrieval-Augmented Generation (RAG) patterns;
3. **Inference of the LLM Model**: at this point the agent sends the entire context (system messages, conversation history and information from memory) to the LLM client to generate a response. During this phase the LLM processes the information and may produce an internal 'thought' which, if present, is captured and recorded as an event. Execution may also take place in streaming mode, providing partial updates of the response in real time;
4. **Model Output Processing and Actions**
    - **Direct Response**: if the LLM generates a textual response or structured output without invoking tools, the agent encapsulates this content in the final response;
    - **Function Calling**: if the LLM decides to invoke one or more tools, the agent manages their execution registering the request as an event and executing the tools asynchronously. The resulting tool output is subsequently added back into the context;
    - **Handoff**: a special case of tool call is the handoff, whereby the agent delegates a task to another agent by sending a message with the relevant context and terminating its execution for that turn;
    - **Reflection on the Use of Tools**: after the execution of the tools - and in the absence of a handoff -, the agent may optionally perform a 'reflection' on the results of the

tools. Indeed, when the relevant reflection parameter is configured, the LLM performs a second inference using the tool results to generate a more refined and coherent response; otherwise, if reflection is not enabled, the agent produces a concise summary of the tool results.

At the end of this cycle, the agent returns a result that includes all internally generated messages and the final response. The termination of the execution can take place according to predefined conditions, providing flexible control over the flow of the conversation.

4.2.3 Multi-Agent Capability

The architectures of the individual agents are combined to form multi-agent systems. In particular, the framework provides sophisticated mechanisms at both the Core and AgentChat levels to orchestrate collaboration between multiple agents.

AutoGen Core supports several multi-agent interaction patterns:

- **Mixture of Agents** is a model inspired by feed-forward neural networks, with worker agents organised in layers and an orchestrator agent;
- **Sequential Workflow** consists of agents performing specific tasks in a deterministic sequence, with each one passing output to the next agent;
- **Group Chat** is managed by a GroupChatManager that selects the next agent to speak, useful for dynamically breaking down complex tasks;
- **Handoffs** is the mechanism in which an agent delegates tasks to other agents via a special tool call, and is scalable in distributed environments;
- **Multi-Agent Debate** is a pattern in which solving agents exchange and refine answers, through an aggregator agent that manages the process;
- **Reflection** is concerned as a pair of agents - or LLM inferences - in which one generates an output and the other critically analyses or iteratively refines it.

AutoGen AgentChat builds on the above concepts by offering higher-level abstractions for agent teams via:

- Predefined **Teams** characterised by classes such as **RoundRobinGroupChat** in which agents speak following a certain turn order, **SelectorGroupChat** in which an LLM selects the next speaker to be passed on, **MagenticOneGroupChat** in which a main orchestrator directs other agents, and **Swarm** which is based on handoffs. All these teams encapsulate common coordination logics;
- Orchestration and **Workflow** so that in addition to teams, AgentChat supports **GraphFlow**, which allows more complex multi-agent workflows to be built based on direct graphs by explicitly defining the execution flow between agents.

Communication between agents takes place primarily through the exchange of messages, the order and flow of which are defined by the interaction pattern or team configuration.

Table 4.1 summarises the main features of the AutoGen architecture and the engineering implications, further detailed in Section 4.6.

Table 4.1: Summary of AutoGen Architecture Features

Characteristic	**Description**
Agent Model	**Core Level:** RoutedAgent (actor model based, event-driven, 'unopinionated' model). **AgentChat Level:** ChatAgent/ConversableAgent (high-level abstraction for 'conversation programming')
	Continued on next page

Table 4.1 – continued from previous page

Characteristic	Description
Key Architectural Components	LLM (reasoning/response) Tools (external actions/functions via FunctionTool, PythonCodeExecutionTool, GraphRAG, HttpTool, MCP) Human Input (via UserProxyAgent) Memory (conversation history, Memory protocol for RAG and data structures, BufferedChatCompletionContext, TokenLimitedChatCompletionContext)
Operating Cycle	Event-driven (Core), driven by a conversational flow (AgentChat)
Agent Programming	"Conversation Programming": definition of roles, capabilities (LLM, tool, human) and message interaction logic
Internal State Management	Stateful, message history, save_state/load_state
Multi-Agent Interaction	Message exchange, predefined patterns (GroupChat, Debate, Handoffs, GraphFlow, MagenticOne), predefined mechanisms (RoundRobinGroupChat, SelectorGroupChat, Swarm)
LLM Stochasticity Management	Mitigated by Tool Integration, Reflection, Multi-Agent Debate, Structured Output
LLM Context Window Management	Mitigated by BufferedChatCompletionContext, TokenLimitedChatCompletionContext, Memory module and RAG
LLM Hallucinations Management	Mitigated by Tool Integration, Reflection, Multi-Agent Debate, Structured Output
LLM Planning Management	Facilitated by conversational decomposition patterns such as Group Chat, Handoffs and GraphFlow
Strengths (Agent Architecture)	Dual architecture to balance power (Core) and simplicity (AgentChat), flexibility, intuitive conversational model, human-in-the-loop integration, multi-agent support, distributed scalability, advanced observability
Potential Weaknesses (Agent Architecture)	Complexity in handling very long conversational states, potential overhead for large-scale scenarios, questionable suitability of the conversational paradigm for non-dialogic tasks, distributed state management
Practical Aspects	**Complexity:** Medium-high for deep customisation **Debugging/Observability:** Elevated through advanced logging and OpenTelemetry **Transparency:** High, given the focus on explicit interaction patterns **Modularity/Reusability:** Very high, with agents and tools as reusable units **Development Time:** Reduced for rapid prototyping using predefined components **Maintainability:** High, thanks to modularity and observability tools

4.3 CAMEL: The Role-Driven Agent and 'Inception Prompting' for Autonomous Co-operation

4.3.1 Vision and Architecture of the Framework

CAMEL (Communicative Agents for MEtacognitive Learning) is proposed as a framework for building LLM agents capable of autonomous cooperation for solving complex tasks, with the primary objective of minimising human supervision. The central vision for defining the agent architecture and behaviour is based on two key principles:

- **Inception Prompting**: is an advanced prompt engineering technique that arose as a direct response to the practical challenges encountered by researchers during preliminary experiments such as 'role flipping' (role reversal between agents) and 'flake replies' (vague responses), which undermined autonomous cooperation. In fact, it is used to initialise agents with extremely precise and structured definitions of their goals, roles, behavioural constraints and communication protocols, with the intention of deeply 'grafting' the desired behaviour into the agent from its creation, going beyond the simple subdivision into Task Specifier Prompt, Assistant Prompt and User Prompt and including mechanisms for interaction control, output validation and expectation management;
- **Role-Playing**: agents are assigned specific roles (e.g. "Python Programmer", "Critical Thinker", etc.). Once these roles have been defined, they talk to each other autonomously, contributing from the perspective of their role, to complete complex tasks. This approach stimulates different cognitive perspectives and specialisations within a collaborative context.

These principles aim to structure the internal architecture of the agent and guide its decision-making process in a more predictable and goal-aligned manner.

CAMEL's architecture is modular and is built around a concept of a basic agent, later extended by more specialised agents and supported by a series of modules dedicated to specific functionalities. In addition, it is enriched by a series of Specialised Modules that provide functionality such as: **Data Generation**, which includes tools for generating high-quality training data, such as Chain of Thought (CoT) with advanced algorithms (MCTS, Binary Search Error Detection), Self-Instruct for creating instruction-related data, Source2Synth and Self-Improving CoT for iterative refinement of chains of reasoning; **Embeddings**, the support for creating vector representations for different types of data (text, images, video); **Memory and Storage**, dedicated modules for short and long-term memory management, with retrieval techniques based on vectors or keywords, and integration with various database systems (SQLite, Redis, Pinecone, Chroma); **Retrievers Module**, an advanced information retrieval system with different types of retrievers and search strategies; **Loaders**, modules for handling various file formats and unstructured data; **Tools and Toolkits**, provide tools (individual functions) and toolkits e.g. collections of tools for specific tasks, e.g. Arxiv Toolkit, CodeExecutionToolkit with support for different sandboxes such as Docker and e2b; **Prompting**, a module dedicated to prompt management, with predefined templates, dictionaries for various tasks and support for the creation of customised prompts; **Fine-tuning and Customisation**, tools to adapt agents to specific domains through techniques such as RLHF (Reinforcement Learning from Human Feedback) and distillation; **Evaluation Systems (CRAB - CAMEL Reliable Agent Benchmark)**, a framework for evaluating agent performance; and **Monitoring and Debugging (AgentOps)**, integrations with external services to track agent execution.
The presence of these specialised modules, in particular those for data generation, fine-tuning and evaluation (CRAB), suggests that CAMEL is conceived not only as a framework but also as a more holistic platform oriented to support the entire life cycle of research and development of this type of agent, positioning it strongly in the academic and experimental sphere. Indeed, its evolutionary trajectory seems to be more focused on enriching this research ecosystem, rather than on integration with enterprise standards.

4.3.2 Agent Architecture

The architecture of a CAMEL agent is defined by its base class and the modules that extend its capabilities.

Definition and Role

The agent's behaviour is primarily defined by the role assigned to it via Inception Prompting. The BaseAgent class imposes fundamental methods such as reset() - to return to the initial state - and step(input) - to perform a single processing step given the current input. The ChatAgent represents the standard implementation that allows us to define an agent for LLM-based conversations that can manage conversation memory, invoke external tools, produce structured output (validated through Pydantic schemes), operate asynchronously and even utilise multiple LLMs through scheduling strategies.

Key Components

LLM Integration and Conversational Memory Management

The ChatAgent integrates an LLM as a reasoning engine and manages the conversation memory, which includes a configurable context window. It also allows more than one LLM to be defined for use within it, which are then managed via a manager by specifying the scheduling strategy.

Using Tools and Structured Output

Agents can invoke external tools (organised in Toolkits) and produce structured output (validated through Pydantic schemes), guaranteeing predictable and readable answers in a certain format. CAMEL also supports the Model Context Protocol (MCP) to expose toolkits as stand-alone servers, facilitating interoperability.

Persistent Memory and RAG

In addition to conversational memory, CAMEL agents can use more persistent memory modules (with support for token limits and integration with databases such as SQLite, Redis, Pinecone, Chroma) and dedicated Retrievers modules for Retrieval-Augmented Generation (RAG). This significantly extends the agent's knowledge base beyond its training data (compared to the LLM) or immediate context.

Specialised Agents

The framework includes variations of the basic agent architecture (BaseAgent or ChatAgent), such as CriticAgent that can be used for evaluation, DeductiveReasonerAgent for logical problem decomposition, EmbodiedAgent for handling sensory/environmental contexts, KnowledgeGraphAgent for interacting with knowledge graphs and TaskAgent for task decomposition, each bringing specific expertise to the system.

Human-in-the-Loop

CAMEL provides mechanisms to integrate feedback and human interaction into the decision-making process of agents via HumanLayer, thus enabling feedback requests, questions and approvals to be handled prior to critical actions. There is also the Critic-in-the-loop pattern where a critical agent (e.g. CriticAgent) evaluates the responses of other agents.

Operating Cycle

The step and astep methods implement the agent's operational cycle in synchronous and asynchronous versions, following a structured iterative pattern. The process begins with the acquisition and storage of the user message converted into a BaseMessage object, followed by the retrieval of the conversational context and sending to the backend language model.

The system implements an advanced mechanism for handling calls to tools through a controlled loop: it distinguishes between internal, which are executed immediately with automatic recording of the results in memory; and external, which interrupt the loop by delegation to the caller. The process incorporates security controls that include token limit monitoring, maximum iteration constraints to prevent infinite loops, and management of external termination events.

For life-cycle control, the reset method provides the restoration of the agent's initial state: it resets the termination flag, re-initialises the memory via init_messages (deleting the history and retaining only the system message) and restarts all registered terminators, ensuring a clean restart for new conversations.

The cycle concludes with the formatting of the response according to any required structured patterns and the return of an object containing the generated messages, diagnostic information on the tokens used, records of tool calls and the agent's termination status.

4.3.3 Multi-Agent Capability

CAMEL places a strong emphasis on multi-agent capabilities, offering specific modules and architectures for coordination and collaboration.
The **Society** module is designed to simulate complex social interactions between agents and includes implementations of autonomous collaboration frameworks such as: **RolePlaying**, characterised by agents with distinct roles defined through 'Inception Prompting' who exchange instructions and responses in turn, collaborating to achieve a common goal and maintaining alignment with user intentions; **BabyAGI**, an implementation of the well-known framework for task-driven autonomous agents.
The **Workforce** module, on the other hand, implements a multi-level hierarchical architecture for solving complex tasks and typically includes: a **co-ordinator**, who assigns tasks to work nodes; a **planner/scheduler**, which breaks down complex tasks into more manageable subtasks; and **worker nodes** that perform subtasks and in which each node can host one or more specialised agents.

Communication within the workforce usually takes place via a shared task channel (pub/sub), which facilitates scalability and role specialisation.
Also an implementation of **TaskAgent** is considered and represents an agent specialised in task decomposition and management, which can be used in both single-agent and multi-agent contexts.
OASIS (Open Autonomous Agent Society Interactive Simulator), which is a component designed for large-scale simulation - potentially with millions of interacting agents: it provides an infrastructure to create complex virtual environments where agents can communicate, collaborate and compete, enabling the study of emergent behaviour.

Communication between agents in CAMEL is structured according to the coordination model adopted. For instance, RolePlaying is based on a shift dialogue, whereas in Workforce there is a centralised publish/subscribe channel that manages the distribution of tasks and information.

This heterogeneity in approaches to collaboration - from the fluidity of dialogue in RolePlaying to the hierarchical structure of the Workforce - reflects a philosophical divergence from frameworks such as AutoGen, which favour a more uniform conversational paradigm. CAMEL seems to offer a range of options from more free (but still role-driven) interactions to more rigid and predefined organisational structures, depending on the nature of the problem and the degree of control desired.

Table 4.2 summarises the main features of the CAMEL architecture and the engineering implications, further detailed in Section 4.6.

Table 4.2: Summary of CAMEL Architecture Features

Characteristic	**Description**
Philosophical Orientation Primary Target	Holistic platform for scientific research and reproducible experimentation on cooperative and metacognitive agent behaviour
Agent Model	Role-driven agent whose behaviour is initialised via 'Inception Prompting'
Key Architectural Components	BaseAgent (reset(), step()) ChatAgent (LLM, conversational memory, tool, structured output) Specialised Modules (Persistent Memory, Retrievers, Toolkits, Fine-tuning, Data Generation, Embeddings, Loaders, Prompting, CRAB Evaluation, AgentOps Monitoring)
Operating Cycle	Iterative via step(input) method; reasoning strongly influenced by role and initial prompt
Agent Programming	Definition of roles, objectives, constraints and communication protocols through "Inception Prompting
Internal State Management	Status managed by BaseAgent/ChatAgent, conversational memory, persistent memory modules (with DB support such as SQLite, Redis, Pinecone, Chroma)

Continued on next page

Table 4.2 – continued from previous page

Characteristic	Description
Multi-Agent Interaction	Role-based dialogue (Society Module with RolePlaying, BabyAGI), hierarchical architecture (Workforce Module with Coordinator, Planner, Worker Nodes), pub/sub channel, OASIS for large-scale simulations
LLM Stochasticity Management	Mitigated by Prompting, Role-playing, Chain of Thought, Self-Improving CoT, Critic-in-the-loop, Structured Output
LLM Context Window Management	Mitigated by configurable context window, short/long-term memories, RAG, consideration of token limits
LLM Hallucinations Management	Mitigated by Chain of Thought (CoT, MCTS, Binary Search Error Detection, Dual-Agent Verification System), Self-Improving CoT, CRAB, Critic-in-the-loop, Structured Output
LLM Planning Management	Facilitated by multi-agent orchestration modules (Society, Workforce), TaskAgent, planner component, Task concept
Strengths (Agent Architecture)	Strong behavioural guidance through roles and 'Inception Prompting', modularity and wealth of integrable capabilities, advanced multi-agent coordination patterns, integrated evaluation and monitoring tools
Potential Weaknesses (Agent Architecture)	Complexity of Inception Prompting, potential rigidity of roles, steeper learning curve, potential dependence on external services (AgentOps)
Practical Aspects	**Complexity:** High, given the wealth of features **Debugging/Observability:** High thanks to integration with AgentOps **Transparency:** Good, thanks to detailed logging and Critic-in-the-loop **Modularity/Reusability:** Very high, with specialised agents and modules **Development Time:** Potential reduction through prebuilt components **Maintainability:** High, thanks to modular architecture and debugging tools

4.4 Eclipse LMOS Arc: DSL-Defined Agent in an 'Operating System for LLM'

4.4.1 Vision and Architecture of the Framework

Eclipse LMOS (Language Model Operating System) aspires to act as an 'operating system' for language models and agents, with the aim of abstracting the complexity of agent development and promoting an interoperable 'Internet of Agents'. In this context, the agent is an entity whose capabilities and logic are defined through a Domain Specific Language (DSL) and which operates within a robust and standardised infrastructure.

The LMOS vision is based on the evolution of the Social Web and the Internet of Things (IoT), extending its principles to web-based multi-agent systems. Guiding principles include openness - i.e. the use of open standards and protocols to eliminate vendor lock-in -, interoperability to ensure communication between agents even from different platforms and ecosystems, transparency, accountability, privacy and data security.

The overall architecture of Eclipse LMOS is layered and reflects its enterprise focus and operating system vision:

- **LMOS Protocol**, defines the foundations for interoperability in the 'Internet of Agents', including discovery mechanisms, a description format for agents and tools, a semantic data model and standards for identity and security. It is based on the W3C Web of Things (WoT) architecture to simplify the discovery and interaction of agents and tools, while remaining transport-independent;

- **LMOS Platform**, a cloud-native platform (based on Kubernetes, Istio, ArgoCD, Argo Rollouts) to build and run enterprise-grade multi-agent systems. It acts as a reference implementation of LMOS Protocol, managing orchestration, scalability and agent security;
- **LMOS Agent ReaCtor (ARC)**, is the development framework in Kotlin (with Java support via Spring Boot) that provides the DSL for defining LMOS agents. ARC is designed to abstract away the complexities of interaction with LLMs, memory management and tool integration, facilitating rapid prototyping and collaboration between developers of different types.

4.4.2 Agent Architecture

The LMOS ARC framework allows the agent to be implemented through a set of DSL constructs that specify its capabilities and behaviour.

Definition and Role
An ARC agent is defined through the DSL, which allows its role and objectives to be specified, often also through the technique of 'Use Case Prompting'. A core design principle of ARC is that each agent must have a single objective or task to complete, from which it must not deviate.

Use Case Prompting is an ARC technique that combines a structured 'use case' format, having certain fields, with prompts and static code to achieve more reliable and predictable agent behaviour. Specifically, the fields concern: the name of the use case; the description, i.e. a detailed explanation of the situation or request; the steps, i.e. a sequence of steps - optional, which can be used to provide additional context or guidance - that the agent must perform before providing the final solution; the solution, i.e. a recommended solution to solve the problem or satisfy the request; the alternative, which can be used if the primary solution is not effective; Fallback, in case the primary and alternative solutions fail; Examples, i.e. a list of example queries or statements that should trigger this use case. Finally, Conditionals are functions that allow certain rows to be omitted from use cases on the basis of certain conditions.

Key Components
LLM Prompt and LLM Integration
The DSL makes it possible to define prompts and interact with various LLM clients (Azure OpenAI, Google Gemini, LangChain4j, Ollama) and to integrate customised clients from other frameworks via the ChatCompleter interface.

Input/Output Filters (filterInput, filterOutput)
The DSL makes it possible to define mechanisms to transform, validate or modify the data exchanged with the agent, influencing its perception and action. Functions such as breakWith allow processing to be terminated and return a dedicated response, while runAsync allows concurrent execution of filters.

Event Handlers (emit())
Agents can proactively issue events to notify other systems or users of significant changes or steps in processing, decoupling communication and providing a way to track progress.

Readers (Knowledge/Perception)
They represent functions callable via DSL to read data from different sources (PDF, HTML), providing dynamic knowledge to the agent.

Functions (Actions/Tool)
They represent external tools, defined with parameters and descriptions that the LLM uses to decide which function to invoke. ARC also supports the integration of tools hosted by MCP servers and can expose its own functions as MCP tools.

Memory Management and Knowledge
It supports persistent and session memory, with In-Memory or MongoDB-based implementations - which can also use TTL indexes for short-term memory. ARC allows an agent to have static knowledge, that which is added directly in the system prompt, or dynamic knowledge that can be injected during execution via Readers and Functions.

Model Extensibility
In ARC, agents can also use other models besides LLMs, such as Computer Vision Models, Speech Recognition Models, Time-Series Models and Recommendation Systems.

Operating Cycle
The ChatAgent component represents the basic entity of the system designed to handle conversational interactions through a well-defined processing pipeline. In particular, the operational cycle is asynchronous, uses Kotlin coroutines and consists of five main phases:

1. **Input Preprocessing**, the agent receives a conversation and applies configurable filters to validate and pre-process the input data;
2. **Dynamic Context Generation**, the system dynamically generates the system prompt based on the current context of the conversation and agent configurations;
3. **Loading and Orchestration of Tools**, the agent dynamically loads the required functions and tools through a dependency injection system;
4. **Invocation of the Linguistic Model**, the central phase consists of the invocation of the configured LLM, which processes the conversation enriched with the available context and tools to generate a semantically appropriate response;
5. **Output Postprocessing and Filtering**, consists of the last phase that applies configurable filters to the generated output, allowing validations, transformations and routing decisions to other agents in the system.

4.4.3 Multi-Agent Capability

LMOS supports multi-agent capabilities at various levels: **LMOS Protocol** and **Platform** are those that enable agent discovery (Agent Registry), communication via open protocols and group management; **LMOS Runtime** and **Router**, on the other hand, orchestrate collaboration and routing. The LMOS Arc DSL provides specific methods, including: callAgent and askAgent to invoke other local or remote agents; nextAgent to pass the conversation to another agent; and constructs such as AgentChain to define predetermined sequences of calls to agents, where the output of one agent represents the input of the next agent in the chain.

Table 4.3 summarises the key features of the Eclipse LMOS Arc architecture and the engineering implications, further detailed in Section 4.6.

Table 4.3: Summary of Eclipse LMOS Arc Architecture Features

Characteristic	**Description**
Philosophical Orientation Primary Target	Development of enterprise-grade multi-agent systems that can be integrated, governed and managed
Agent Model	Entity defined via ARC DSL (Kotlin/Java), operating within the LMOS Platform
Key Architectural Components	Prompt LLM Input/Output Filters (filterInput, filterOutput, breakWith, runAsync) Event Handlers (emit()) Readers (dynamic knowledge from PDF, HTML) Functions (tool, MCP integration) Memory (LONG$_TERM, SHORT_TERM, In-Memory, MongoMemory$) Extensibility of the model (Computer Vision, Speech Recognition, etc.)
Operating Cycle	Execution of the logic defined in the DSL "Use Case Prompting" to structure the behaviour (Name, Description, Steps, Solution, Alternative, Fallback, Examples, Conditionals)

Continued on next page

Table 4.3 – continued from previous page

Characteristic	Description
Agent Programming	Via ARC DSL, combining prompt definitions, Kotlin logic, and interaction with platform components (e.g. access to bean via get())
Internal State Management	LONG_TERM and SHORT_TERM memory, data management via DSL (getData, addData)
Multi-Agent Interaction	DSL constructs (callAgent, askAgent, nextAgent, AgentChain, Calling Remote Agents) supported by LMOS Protocol/Platform (Agent Registry, discovery, routing, Group Management)
LLM Stochasticity Management	Mitigated by Use Case Prompting, Input/Output Filters, Functions (Tools), Model Extensibility
LLM Context Window Management	Mitigated by structured memory management (LONG_TERM, SHORT_TERM)
LLM Hallucinations Management	Mitigated by Use Case Prompting, Input/Output Filters, Functions (Tools)
LLM Planning Management	Facilitated by Use Case Prompting (Steps), AgentChain, callAgent/askAgent by delegation
Strengths (Agent Architecture)	Structured definition (DSL), enterprise robustness and scalability (LMOS Platform with security with Istio mTLS/RBAC, controlled releases with ArgoCD/Rollouts), interoperability (LMOS Protocol, WoT, DIDs), advanced observability, enterprise integration (Spring Boot, GraphQL)
Potential Weaknesses (Agent Architecture)	Dependence on the JVM ecosystem, complexity of the underlying infrastructure (Kubernetes) for simple agents, potentially steep learning curve
Practical Aspects	**Complexity:** Medium-high **Debugging/Transparency:** Very high thanks to tracing, events, Arc View **Modularity/Reusability:** Very high thanks to DSL, bean, functions, remote agent collaboration **Development Time:** Accelerated for rapid prototyping **Maintainability:** High, thanks to standardization and structured management

4.5 AgentLite: A Minimal Agent Architecture

In addition to more structured frameworks, there are approaches such as AgentLite that focus on lightness and ease of experimentation with new agent architectures, which are particularly useful in research contexts.

4.5.1 AgentLite Vision and Architecture

AgentLite is a 'lightweight' and 'user-friendly' open-source Python library with framework-like architectural principles, designed to simplify the creation, evaluation and innovation in the field of task-oriented generative agents. Its vision is to provide the basic building blocks for a minimal agent architecture, allowing developers to focus on specific innovation (e.g. reasoning strategies, memory mechanisms) rather than on managing a complex framework.

The architecture of an AgentLite agent is deliberately minimal, centred around the BaseAgent class and its essential modules, which together constitute an explicit and modular agent model.

Definition and Role
A BaseAgent inherits basic functionality to interact with the environment and perform tasks. Its role is typically defined by a descriptive string.

Key Agent Architectural Components
PromptGen
It consists of the module responsible for the dynamic construction of the prompt sent to the LLM: it assembles the role description, task instruction, constraints, available actions and memory of past interactions.
Actions
They represent the operational capabilities of the agent (tool). Developers may create subclasses of BaseAction, specifying action_name, action_desc (description for the LLM) and params_doc; while the call() method is the one that executes the action and returns an observation.
LLM
It is the module representing the interface to the language model (e.g. BaseLLM wrapper for OpenAI), which receives the prompt from PromptGen and returns a textual response (thought or action).
Memory
Represents the module for storing the historical sequence of actions performed and observations received. This history is then used by PromptGen to provide context to the LLM.

Operating Cycle
A typical AgentLite agent workflow is iterative and follows this sequence:

1. Receiving a Task Instruction (TaskPackage);
2. PromptGen creates a prompt using the instruction and Memory;
3. The prompt is sent to the LLM;
4. The LLM generates a response (action to be performed or thought, e.g. for ReAct strategies);
5. The agent performs the action specified via the Actions module;
6. The environment/tool returns an observation;
7. Action and observation are stored in Memory;
8. The process is repeated until the task is completed (e.g. action 'Finish').

This modular architecture, albeit minimal, reflects an emerging consensus on the fundamental components of a task-oriented generative agent.

4.5.2 Multi-Agent Capability

AgentLite supports the creation of multi-agent systems, typically orchestrated by a ManagerAgent (subclass of BaseAgent, with enhanced communication capabilities), which coordinates the activities of other individual agents often in a hierarchical structure. Each subordinate agent can be seen as a specialised 'action' for the manager agent.

4.5.3 Distinctive Positioning, Analysis and Implications

AgentLite's minimalist philosophy, which breaks down the agent into essential components such as PromptGen, Actions, LLM and Memory, provides an opportunity for research into the fundamentals of LLM-based agenticity. This modular approach makes it possible to isolate and study specific aspects of the agent lifecycle, such as the management of 'beliefs' (via Memory) or the execution of 'actions' (via Actions), in a similar way to how the components of the BDI cycle are studied and implemented in the relevant agents.

Thus, although AgentLite does not impose an explicit cognitive model like the BDI, its architecture facilitates the experimentation of reasoning strategies that emulate or are inspired by deliberative processes. In this sense, it also serves as a 'laboratory' for exploring how to engineer 'deliberation' into a generative agent, allowing different implementations of the reasoning-action cycle to be tested and potentially bridging the conceptual gap between symbolic and generative agents.

Research and Innovation vs. Enterprise Functionality

AgentLite is designed to facilitate research and innovation in generative agent behaviour through its minimalist architecture and modular components (e.g. PromptGen, Actions, LLM, Memory) that enable rapid prototyping of new reasoning strategies (e.g. ReAct, Reflection) and agent architectures, focusing on module-specific innovation without having to manage a complex framework. Its simplicity and modularity make it easy to experiment with different implementations of each component.

This research orientation implies less emphasis on enterprise-level 'out-of-the-box' functionalities, such as large-scale deployment, high availability or advanced resource management. The choice of AgentLite represents therefore a compromise between maximum flexibility for experimentation and the robustness/completeness required for complex production applications.

LLM Stochasticity Management and Predictability Improvement

AgentLite addresses the limitations of LLM through its modular architecture, which allows for the integration and experimentation of various strategies:

- **Context Management**
  The Memory module explicitly manages the history of actions and observations, used by PromptGen to provide relevant context to the LLM, helping to maintain consistency across multiple shifts and mitigating the limited context window of LLMs;

- **Reasoning Strategies**
  AgentLite is designed to facilitate the rapid prototyping of new reasoning strategies, such as ReAct (Reasoning and Acting) and Reflection, thus enabling the customisation of the PromptGen and Actions modules, the implementation of reasoning flows that guide the output and decision-making process of the LLM, going beyond the simple response to the prompt;

- **Use of Tools**
  The Actions module provides a clear mechanism for the integration of external tools, which is crucial for 'grounding' LLM agents in reality and overcoming the inherent knowledge limitations and potential 'hallucinations' of LLMs.

Although AgentLite does not provide complex predefined solutions, its modular flexibility allows researchers to develop and test innovative approaches to try to make the behaviour of LLM more structured and predictable.

Multi-Agent Coordination

The AgentLite approach offers a flexible way to build collaborative systems without imposing overly prescriptive communication protocols, allowing different coordination patterns depending on the task. However, highly structured or complex multi-agent coordination - needed for instance in enterprise applications - may require more extensive custom development than frameworks that offer higher-level abstractions for such scenarios.

Practical Aspects

Perceived Complexity
AgentLite is designed to be light and simple, with clear basic components that are easy to extend. In this way, it reduces the overhead typically associated with more complex frameworks.

Debugging and Transparency
Although no specific details have been officially provided, the modular nature and clear operating cycle of the agent facilitate understanding of the execution flow, which is suitable for debugging. In addition, transparency is intrinsic to the clarity of the components.

Modularity and Reusability
The architecture is inherently modular, with PromptGen, Actions, LLM and Memory as discrete, modular modules, which promotes high reusability of individual components and the possibility to easily exchange or extend them.

Development Time
Simplicity and modularity are designed for rapid prototyping, reducing development time for experimentation.

Maintainability
The clarity of the components and their interactions contributes to good maintainability, as changes to a specific module have little impact on the rest of the system.

Table 4.4 summarises the key features of the AgentLite agent architecture and the engineering implications discussed.

Table 4.4: Summary of AgentLite Architecture Features

Characteristic	**Description**
Agent Model	Minimal and modular architecture, designed as a 'meta-framework' for research and rapid prototyping of new architectural patterns for LLM-based agents
Key Architectural Components	PromptGen (prompt construction) Actions (executable tools) LLM (model interface) Memory (action-observation history)
Operating Cycle	Instruction -> PromptGen -> LLM -> Actions -> Memory. Iterative to completion
Agent Programming	Role definition, implementation of custom Actions, configuration of PromptGen and Memory
Internal State Management	Via Memory module (action-observation history)
Multi-Agent Interaction	Orchestration via 'manager agent', hierarchical structure, less prescriptive communication
LLM Stochasticity Management	Mitigated by modularity for experimentation with reasoning strategies (e.g. ReAct, Reflection)
LLM Context Window Management	Mitigated by Memory module for explicit context management
LLM Hallucinations Management	Mitigated by Actions module for tool use and reasoning strategies
LLM Planning Management	Facilitated by iterative operating cycle and implementable reasoning strategies
Strengths (Agent Architecture)	Serves as a 'laboratory' for testing individual components of agentivity (e.g. memory, reasoning, action selection)

Continued on next page

Table 4.4 – continued from previous page

Characteristic	Description
Potential Weaknesses (Agent Architecture)	Less out-of-the-box support for complex enterprise functionality or highly structured multi-agent interaction patterns, less emphasis on advanced scalability and observability
Practical Aspects	**Complexity:** Low for basic uses, medium for complex scenarios **Debugging/Transparency:** Good, given the clarity of the components **Modularity/Reusability:** Very high, thanks to modular design **Development Time:** Very small for prototyping **Maintainability:** High, thanks to simplicity and modularity

4.6 Critical Comparative Analysis

The descriptive analysis of the frameworks revealed considerable convergence at the level of architectural components. However, it is in the analysis of their engineering philosophies and implementation choices that the dual thesis of this work emerges: a "functional convergence" towards AOSE problems, coupled with a clear "paradigmatic divergence" in implementation solutions.

4.6.1 Comparison of Control Strategies: Flexibility vs. Predictability

Each framework offers a different engineering trade-off between the emerging flexibility of LLM and the need for deterministic control.

AutoGen: Flexibility vs. Control in Conversation Programming

AutoGen's 'conversation programming' paradigm offers a high degree of flexibility by allowing the definition of dynamic interactions in which agents fluidly adapt their strategies according to the ongoing dialogue. While this promotes emergent behaviour and less rigid workflows, ideal for rapid prototyping and open problem solving, it also raises an important issue: the potential cost in terms of deterministic control and predictability. This points to the core engineering choice of the framework: deterministic control, typical of formal models, is replaced here by emerging and flexible coordination as the primary governance mechanism. The paradigm's core hypothesis is that the dialogue itself can become a self-correction mechanism, capable of mitigating the very stochasticity it introduces.
The framework's authors counter this by presenting empirical evidence to support the power of this paradigm precisely in scenarios where rigidity would be a limitation. In applications like Conversational Chess and Dynamic Group Chat, the ability of agents to dynamically orchestrate turns of phrase and adapt strategy proves crucial to success. In these cases, flexibility is not a weakness but an emerging coordination mechanism that can be more efficient than a static workflow.
The strongest evidence lies in the Math Problem Solving task, where a two-agent dialoguing system achieved an accuracy of 69.48% on the MATH benchmark, outperforming more rigid single-agent approaches such as ChatGPT+Code Interpreter. This result suggests that the LLM's stochasticity is effectively mitigated by conversational patterns like 'proposal-check-correction', validating the dialogue's role as a robustness-enhancing mechanism.
Communication itself, although message-based as in classical MAS architectures, evolves from formal protocols to a governed conversational flow, which proves to be a valuable tool for adaptability and robustness.

CAMEL: Structure and Flexibility through "Inception Prompting"

CAMEL's strong reliance on 'Inception Prompting' and 'Role-playing' directly addresses the inherent stochasticity of LLMs by imposing significant behavioural structure and constraints. This approach aims at a much more predictable and aligned agent behaviour by channelling the LLM's generative capabilities into pre-defined and specific roles and interaction patterns.

As reported in the authors' experiments, the effectiveness of this highly structured approach was validated through both human and automatic (with GPT-4) evaluations. Indeed, when comparing

solutions to complex tasks generated via CAMEL's role-playing dialogue with those produced by a single LLM (gpt-3.5-turbo), CAMEL's solutions were found to be superior in 76.3% of the cases by the human evaluators and 73% of the cases by GPT-4 for the tasks in the 'AI Society' dataset, with similar results for the 'Code' dataset, empirically demonstrating that the decomposition of the problem into specialised roles, guided by rigorous prompts, produces qualitatively better results, and justifying the engineering trade-off between the rigidity of the framework and the performance of the final solution.

Thus, this strong structuring makes CAMEL particularly effective for tasks requiring collaborative problem solving within well-defined domains and for the generation of predictable outputs. It also significantly reduces ambiguity in LLM responses and guides generation towards desired outcomes, mitigating randomness and potential hallucinations. The system's ability to 'tame' the inherent unpredictability of the LLM is a direct consequence of these design choices.
The strong behavioural guidance provided by 'Inception Prompting' comes at a cost in terms of reduced flexibility for highly open-ended exploration or the generation of responses outside predefined boundaries. There is a potential risk of 'over-binding' the LLM, which could make it too rigid to adapt to unforeseen borderline cases or dynamic changes in the task environment.

Designing effective 'Inception Prompts' and defining clear, non-overlapping roles for complex tasks is inherently challenging and requires significant iterative refinement and a deep understanding of prompt engineering principles. This can increase the initial prompt engineering effort and setup time compared to frameworks with less prescriptive control mechanisms.

Eclipse LMOS Arc: Control, Predictability and Flexibility via DSL

The use of a specific DSL for agent definition in LMOS Arc gives a high degree of control and predictability over agent behaviour. The structuring of prompts through 'Use Case Prompting' and the possibility of defining Conditionals within use cases allow the LLM to be guided towards precise responses and actions, potentially reducing ambiguity and stochasticity. In addition, the presence of input/output filters allows for more granular control over the flow of information.

This strong emphasis on predictability and control makes LMOS Arc particularly suitable for enterprise contexts where robustness, reliability and compliance with complex business logic are priorities. Indeed, its evolutionary trajectory focused on the adoption of cloud-native standards and open protocols positions it as a solution for long-term integration into enterprise ecosystems. However, the inherent rigidity of a DSL and use-case approach may limit flexibility for highly exploratory tasks or scenarios requiring non-default adaptability.

Flexibility, on the other hand, is achieved through the ability to integrate several LLM clients, Readers for various data sources, extensible functions (tools) and the possibility to collaborate with remote agents or other frameworks. Integration with Spring Boot and a GraphQL API add further flexibility for integration into existing ecosystems.

4.6.2 Common Strategies for LLM Stochasticity Management

Regardless of differing control strategies, all frameworks converge on adopting a similar engineering "toolkit" to mitigate the inherent limitations of LLMs.

Integration of Tools and External Grounding

By allowing LLMs to invoke external tools, AutoGen and CAMEL root agent responses in factual data and real-world actions. This directly mitigates hallucinations by providing external data sources and capabilities beyond the LLM's inherent knowledge. Indeed, the effectiveness of this integration is quantified in several applications mentioned by the authors such as in the Retrieval-Augmented Chat system (AutoGen), where the introduction of an agent capable of retrieving documents from an external knowledge base significantly improves performance on Question Answering tasks, as demonstrated on the Natural Questions dataset. Or, even more evident is the case of Multi-Agent Coding based on OptiGuide in which the multi-agent architecture with specialised roles - a Writer to write the code and a Safeguard to validate it - led to a 35% F1-score increase with GPT-3.5 in identifying unsafe code, demonstrating how the decomposition of capabilities into distinct agent-tools increases the robustness of the overall system.
Functions (Tools), in Eclipse LMOS Arc, allow agents to access external services and data, 'anchoring' the LLM's responses to reality and overcoming the limits of its intrinsic knowledge and positively impacting in reducing hallucinations and enabling concrete actions.
Also, Eclipse LMOS Arc differs for the Model Extensibility, the ability to integrate models other

than LLMs (computer vision, speech recognition, etc.) that enables the construction of hybrid agents that exploit the strengths of different AIs for specific tasks, overcoming and/or augmenting some of the inherent limitations of LLMs alone.

Structured Reasoning, Reflection, and Output Control

This design pattern in AutoGen allows agents to engage in self-criticism and to iteratively refine their outputs, which helps to identify and correct errors or inconsistencies that might arise from initial LLM hallucinations.
By simulating multi-turn interactions - via the AutoGen multi-agent debate - in which agents exchange and refine answers based on each other's input, this pattern facilitates collaborative error detection and refinement leading to more robust and accurate solutions and reducing the likelihood of a hallucinated answer being presented as final - such as in the Math Problem Solving case mentioned earlier.
Also, the ability in AutoGen to force LLMs to return structured output like JSON text with predefined patterns - e.g. using Pydantic BaseModel classes - inherently reduces the generation of unstructured or meaningless 'hallucinated' text and helps to incorporate Chain-of-Thought reasoning into agent responses.
CAMEL addresses hallucinations by using **Chain of Thought** (CoT), which generates and validates explicit reasoning paths, using advanced algorithms such as Monte Carlo Tree Search (MCTS), Binary Search Error Detection and Dual-Agent Verification System to improve accuracy and verifiability of outputs; **Self-Improving CoT**, which iteratively refines Chain-of-Thoughts through self-criticism allowing agents to identify and correct errors; **CRAB**, the evaluation framework that helps identify and address hallucinations through effectiveness and consistency metrics; **Critic-in-the-loop**, which introduces a critical agent to select proposals and provide continuous feedback, trying to reduce the generation of misinformation; and **Structured Output**, with the use of Pydantic schemas for output ensuring that information is presented in a defined format and less subject to arbitrary text.
In Eclipse LMOS Arc, the Use Case Prompting structures agent behaviour by combining prompts with static code and logic based on use cases, making responses more reliable and predictable and tending to reduce the inherent uncertainty of LLMs.
On the other hand, the Eclipse LMOS Arc' Input/Output Filters allow the transformation, validation or modification of data exchanged with the agent, influencing its perception and action. This helps to manage and control the output of the LLM, mitigating hallucinations and ensuring consistency.

Context and Memory Management

Features in AutoGen such as BufferedChatCompletionContext and TokenLimitedChatCompletionContext are crucial to effectively manage the LLM context window, thus improving consistency and continuity in long conversations.
CAMEL's ChatAgent manages a configurable context window and supports short and long term memories (vector or keyword), integrating RAG to extend agent knowledge beyond the immediate context window.
Regarding the structured memory of Eclipse LMOS Arc, the distinction between short- and long-term memory (SHORT_TERM, LONG_TERM) and different implementations (In-Memory, Mongo Memory) allow agents to maintain a consistent context and retrieve relevant information, overcoming some of the limitations of the context window of LLMs.

Planning and Task Decomposition

AutoGen facilitates planning through various multi-agent collaboration patterns, including Group Chat, Handoffs (for task delegation) and GraphFlow (for direct workflows). These provide structured ways for agents to dynamically decompose tasks and orchestrate complex workflows, directly addressing the scheduling limitations of LLMs.
CAMEL enables improved complex planning through multi-agent orchestration modules such as the **Society** module (with RolePlaying and BabyAGI) and the **Workforce** module (with hierarchical architecture of coordinators, planners and workers) that allow decomposition of complex tasks and autonomous collaboration; the introduction of the concept of **Task** as a specific assignment that can be delegated and broken down into subtasks; **TaskAgent** and the **Planner** component, specifically designed to break down complex tasks into simpler steps.

Synthesis of Mitigation Strategies

Collectively, these strategies—spanning tool integration, structured reasoning, memory management, and task decomposition—demonstrate a clear engineering convergence. Although their specific implementations differ (e.g. conversational patterns, hierarchical role-based modules , or formal DSL constructs), all frameworks leverage these external mechanisms to structure and constrain the LLM's behavior. They effectively 'tame' the inherent randomness of the language model, provide significant control, and thus improve the overall predictability, reliability, and capabilities of the agents they produce.

4.6.3 Comparison on Scalability and Robustness

The manner in which a framework manages these characteristics is a direct consequence of its target audience. As the comparison shows, there is a shift from flexible architectures designed for prototyping to complex infrastructures designed for reliability in production environments:

AutoGen

AutoGen supports both local (single-process) and distributed (multi-process via gRPC) execution, allowing flexible scalability from prototyping to production environments. The underlying event-driven architecture is specifically designed for distributed, scalable and resilient agent systems. In fact, this architectural choice makes AutoGen particularly suitable for complex engineering problems that require distributed processing, high concurrency and the ability for agents to operate on multiple machines or organisational domains.

CAMEL

CAMEL is designed for large-scale simulations through the OASIS component that supports interactions with millions of agents, thus providing an infrastructure for complex virtual environments. In addition, hierarchical structures in the Society and Workforce models help to manage complexity as the number of agents increases.
The presence of feedback management mechanisms such as HumanLayer and the Critic-in-the-loop indicates a focus on validation and reliability of agent interactions, contributing to the overall robustness of the system.

Eclipse LMOS Arc

The LMOS Platform is based on Kubernetes, Istio, ArgoCD and Argo Rollouts, providing a cloud-native architecture for orchestrating and scaling agents. Istio in particular ensures efficient traffic distribution, load balancing, testing of new agent versions (canary deployments) and security via mTLS and RBAC. On the other hand, GitOps via ArgoCD and Argo Rollouts automates deployment and ensures controlled releases. This solid infrastructure makes LMOS Arc suitable for large-scale distributed multi-agent systems in enterprise contexts, where robustness, availability and automated lifecycle management are critical requirements.

4.6.4 Comparison of Practical Aspects

Apart from theoretical capabilities, the adoption of a framework is often also driven by developer experience. A comparative analysis of practical aspects highlights the different trade-offs that each framework imposes on developers in terms of complexity, development time and long-term maintainability:

AutoGen

Perceived Complexity
AutoGen aims to simplify the development of complex agent applications through its 'conversation programming' paradigm and the provision of pre-defined agents and teams. On the other hand, it also provides more flexibility through the Core layer that allows deeper customisation of the definition of agents and their components and behaviours, offering a further implementation alternative with lower-level mechanisms for more experienced users.

Debugging and Observability

AutoGen provides robust support for debugging and agent observability through advanced logging, both textual and structured JSON and native integration with OpenTelemetry for metrics and traces. In this way, full logging and tracing capability becomes essential for diagnosing problems in complex multi-agent systems.

Transparency
The explicit focus on interaction patterns in 'conversation programming', combined with detailed and structured logging, significantly improves the transparency of the agent's behaviour as it allows both message flows and internal events within the system in which the agent is running to be clearly traced.

Modularity and Reusability
Modularity in Autogen is a fundamental principle with agents designed as independent, autonomous and reusable units. In fact, the framework is highly customisable, allowing for customised agents, tools, memory mechanisms and workbenches (collections of tools), all of which favour reusability.

Development Time
Development time is facilitated by the 'conversation programming' approach, the predefined agents (e.g. AssistantAgent, UserProxyAgent) and the pre-configured multi-agent collaboration schemes in AgentChat, which enable rapid prototyping and reduce the need to implement common coordination logics from scratch.

Maintainability
The modular and event-driven architecture, the clear separation of responsibilities - whereby the agents manage their own logic and the framework provides the communication infrastructure - , extensive logging and multi-language support (Python and .NET) contribute positively to the maintainability of AutoGen-based systems.

CAMEL

Perceived Complexity
The large feature set, specialised agents and advanced techniques such as 'Inception Prompting' and multi-agent orchestration, may suggest a steeper initial learning curve and greater perceived complexity for new users. However, modular design and clear abstractions can mitigate this.

Debugging and Monitoring
CAMEL integrates with AgentOps, an external tracking service, to monitor and debug the execution of agents. This provides detailed logs of tool calls, cost metrics and session replays, all crucial for identifying and resolving problems.

Transparency
The detailed logging and session replay capabilities offered by AgentOps contribute to the transparency of agent behaviour. Through it, the developer can observe the inner workings of agents, including the tools invoked and the sequence of actions.

Modularity
CAMEL's architecture is highly modular, with the BaseAgent as the abstraction for all agents, specialised agent classes, and dedicated modules for tools, memory and data retrieval. This modularity promotes reusability and clarity of design.

Development Time
The availability of pre-built specialised agents, tools and toolkits, together with standardised communication protocols, can potentially reduce development time by providing ready-to-use components and established patterns for agent interaction.

Maintainability
The consistent abstraction of the BaseAgent, structured message objects (BaseMessage) and detailed monitoring/debugging capabilities (AgentOps) contribute positively to maintainability. This well-defined architecture with clear interfaces and robust debugging tools facilitates understanding, modification and extension of the system over time.

Eclipse LMOS Arc

Perceived Complexity
The framework attempts to reduce complexity through DSL, abstraction of LLM complexities and integration with Spring Boot. However, the need to understand the overall LMOS Platform architecture (Kubernetes, Istio) and the Kotlin/JVM DSL could make the learning curve medium-high.

Debugging and Transparency
LMOS Arc offers robust support for debugging via Agent Tracing (with Micrometer Tracing exportable to Zipkin, Wavefront, OTLP) and an Arc View GUI that provides a chat UI, event log and performance graphs. The ability to output customised events contributes to transparency and communication decoupling.

Modularity and Reusability
Modularity is promoted by the ability to define agents as Spring beans, the use of reusable functions (tools) and filters, and DSL constructs for collaboration between agents (callAgent, AgentChain, etc.). The ability to call remote agents significantly expands reusability and interoperability in distributed environments.

Development Time
The Kotlin DSL and integration with Spring Boot aim at rapid prototyping and increased productivity, reducing time spent on low-level configurations. The Gradle plugin (experimental) aims to further simplify compilation and deployment.

Maintainability
The use of standardised formats (e.g. JSON-LD for the description of agents/tools), the clear definition of agent goals, the structured management of knowledge (static and dynamic) and memory, all contribute to a higher maintainability of the system over time.

4.7 Synthesis and Convergence on AOSE Principles

The in-depth analysis of the AutoGen, CAMEL, Eclipse LMOS Arc and AgentLite frameworks revealed a remarkable convergence at the level of basic architectural components: an LLM core for reasoning, a memory system and an interface for the use of external tools. More significantly, however, it revealed deep philosophical and engineering divergences in the way these components are orchestrated and conceptualised.

This section argues that the real distinction between frameworks lies not so much in what an agent is, but rather in how its behaviour is programmed, controlled and made reliable. Each framework proposes a different engineering response to the fundamental challenge of imposing structure and predictability on the inherently stochastic behaviour of LLMs, marking a transition from the discovery of necessary components to the design of robust, purposeful systems from those components.

4.7.1 Comparative Analysis of Implemented AOSE Patterns

Analysis of the framework components reveals an alignment, which is often implicit, with the fundamental abstractions of AOSE.

AutoGen

A further key that connects AutoGen's approach to the established principles of AOSE emerges by analysing its components through the lens of the Agents & Artifacts (A&A) model and CArtAgO in which the environment is not a passive background, but an active entity populated by artifacts that mediate and regulate interaction. AutoGen's GroupChatManager can be interpreted as an example of a co-ordination artifact, i.e. a programmable entity shared in the environment whose purpose is to manage a specific interaction rule - in this case, turns of speech. Similarly, the UserProxyAgent acts as an artifact that mediates the interaction between the agent system and the human user, abstracting the complexity of input/output and providing a structured interface. This interpretation reveals how AutoGen, while starting from a conversational paradigm, implicitly implements solutions to coordination problems that AOSE has long formalised through artifact abstraction.

A comparison with the AOSE organisational dimension, such as the Moise model, highlights a clear paradigmatic divergence. Unlike the formal AOSE specifications defined in Chapter 2, AutoGen Teams implement a form of organisation with a different approach. It is not based on a formal separation of structural, functional and deontic dimensions; instead, organisation in AutoGen is implicit and emerges from dialogue, or is defined programmatically, rather than being specified and imposed by an explicit organisational platform as in JaCaMo.

CAMEL

CAMEL's approach, with its emphasis on 'Inception Prompting' and 'Role-playing', can be seen as an attempt to reintroduce a form of structure and predictability into the behaviour of LLM-based agents, recalling in part the principles of AOSE. Indeed, the explicit definition of roles and goals for agents, albeit implemented through advanced prompt engineering rather than formal organisational models, aims to channel agent autonomy towards predefined goals.

It should be pointed out that this approach diverges paradigmatically from formal organisational models such as Moise - often integrated into AOSE platforms such as JaCaMo - because whereas 'roles' in CAMEL represent behavioural guides defined by natural language text, roles in Moise are formal entities embedded in a structural specification - defining hierarchies and relationships between roles -, with explicit obligations and permissions towards collective goals (missions) defined in a functional specification, all governed by a deontential specification.

Instead, the Workforce architecture with its shared publish/subscribe task channel can be seen as implementing a coordination artifact e.g. an environmental mechanism that decouples task producers from consumers, in line with the CArtAgO philosophy. This aligns with the idea of designing multi-agent systems where collective behaviour emerges from the collaboration of specialised agents, although the 'deliberation' mechanism - the role-driven dialogue between LLMs - differs radically from the explicit, programmable deliberative cycle of BDI agents. Thus, 'Inception Prompting' can be interpreted as an engineering strategy to 'pre-fill' the deliberative process by providing a rigid script to guide the LLM's opaque reasoning.

Eclipse LMOS Arc

The adoption of a DSL in Eclipse LMOS Arc for the definition of agents represents an approach that recalls the formalisation and structuring typical of AOSE. The strongest parallelism emerges in comparison with the CArtAgO model, which promotes the environment as a first-class abstraction populated by programmable artefacts.

In LMOS Arc, the Readers and Functions constructs are not simple function calls, but represent the direct counterpart of the CArtAgO artefacts: they are resources exposed in the environment that the agent can observe (Readers) and use (Functions), providing a structured and governed interface to the external world and system capabilities. This architectural choice, together with the DSL, imposes an explicit framework for the agent's logic that contrasts with AutoGen's more fluid 'conversation programming', offering greater deterministic control over agent behaviour that is essential for enterprise applications. In this way, the agent's 'deliberation' does not emerge from a dialogue, but is orchestrated by a formal procedural logic that invokes the LLM as a specialised component, coming closer to the explicit control of BDI systems.

The vision of an 'operating system for LLM' and the emphasis on interoperability through open protocols align with the goal of creating robust and scalable multi-agent environments where agents can interact in a predictable manner, a key aspect in complex multi-agent systems.

4.7.2 Control Strategies and the Challenge of Deliberative Transparency

Based on the analysis of control strategies (section 4.6.1), engineering differences are evident in the various approaches: from emerging control through "Conversation Programming" (AutoGen), to prescriptive guidance through "Inception Prompting" (CAMEL), to formal specification through DSL (LMOS Arc). Each of these approaches addresses the challenge of deliberative transparency in a different way.

As discussed previously (section 3.1.3), these different control approaches do not resolve the inherent challenges of emergent reasoning, such as error propagation and hallucinations. However, the engineering challenges of this approach are the direct consequence of a fundamental architectural feature: the inherently opaque nature of the deliberative process (being based on LLM).

This fundamental difference marks a significant divergence from traditional AOSE paradigms, where agent behaviour is typically defined through agent programming languages (such as AgentSpeak in Jason) that explicitly specify plans, beliefs, and desires.

Unlike BDI systems, which offer deterministic control and a clear traceability of reasoning through an explicit and inspectable deliberative cycle, all the generative frameworks examined take a different approach where the deliberative logic resides in the LLM component, which operates as a 'black box'.
Each framework attempts to "engineer" this opaque process differently: AutoGen uses dialogue flow ('Conversation Programming'), CAMEL uses role pre-conditioning ('Inception Prompting'), and LMOS Arc uses formal logic (DSL).
In all these cases, unlike the BDI cycle where each step - belief update, goal selection, plan execution - is explicit and can be traced, the reasoning of a generative agent must be inferred a posteriori. Features such as advanced logging and tracing, therefore, are not just good engineering practices; they become an essential mechanism for governability, necessary to reconstruct externally the causal chain of the deliberative process which, unlike BDI systems, is not explicit by design.
These control strategies, in an attempt to orchestrate and govern the "black box" of LLM, rely on a common engineering "toolkit" (tool integration, memory management, and RAG) to mitigate stochasticity - as analysed in Section 4.6.2.

4.7.3 The Evolutionary Dynamics of Frameworks: A Race for Continuous Integration

A further critical dimension that emerges from the analysis is that the frameworks examined are not static software artefacts, but dynamic ecosystems in an almost constant state of flux. Their evolution is closely linked to the fast pace of LLM research where any significant advancement in basic research - be it new techniques to improve reasoning, emerging standards for interoperability of tools such as the Model Context Protocol (MCP), or the development of LLMs with more integrated planning capabilities - is promptly taken up and integrated by major frameworks.

This phenomenon is not coincidental, but a direct consequence of the engineering challenges discussed: as LLMs have inherent limitations, each new discovery is seen as a potential engineering 'patch' to mitigate them or 'feature' to enhance their capabilities, and as a result, framework development teams are engaged in a kind of 'race to integration' to remain relevant. This highlights both the relative immaturity of the field and how the 'philosophy' of a framework influences how these new features are integrated, making the evolutionary trajectory of a framework as important as its current state.

4.7.4 Discussion of Common Challenges and Limitations

Despite progress, generative agent architectures face many common challenges, many inherited from the limitations of LLMs themselves:

- **Controllability vs. Autonomy**
  Balancing agent autonomy with the need for alignment and control of its behaviour remains a key challenge;
- **Long-Term Planning and Finite Context Management**
  LLMs' limited context window hinders the agent's extensive planning and long-term consistency;
- **Boundary of Knowledge and 'Hallucinations'**
  The agent may inherit the LLM's tendency to generate incorrect or fabricated information, a critical problem in many contexts;
- **Prompt Robustness and Reliability**
  The sensitivity of LLMs to prompts can affect the reliability of agent behaviour;
- **Efficiency and Costs**
  Interactions with LLMs are computationally intensive and can make architectures slow and expensive;
- **Debugging and Explainability**
  Understanding the 'why' behind the decisions of an LLM-based agent architecture is complex. As discussed in Section 4.7.2, the opaque nature of the LLM's 'black box' reasoning poses a significant engineering challenge for verifiability, which must be addressed using external methods, unlike the intrinsic traceability of BDI systems.

- **Evaluation**
  Defining reliable metrics and benchmarks to evaluate the performance of complex, non-deterministic agent architectures is an open challenge.

These challenges indicate that the field of generative agent engineering is still maturing and that the application of more rigorous software engineering principles and inspiration from classical cognitive architectures could offer promising avenues.

4.7.5 Detailed Critical Comparison Table

Table 4.5 provides a detailed comparative summary of agent architectural patterns and related engineering capabilities among the four frameworks examined, visually encapsulating the divergences and convergences discussed in this section.

Table 4.5: Critical Comparison of Architectural Patterns of Agents and Related Engineering Capabilities

Comparison Criterion	AutoGen	CAMEL	Eclipse LMOS Arc	AgentLite
I. DESIGN AND CONTROL VISION				
Control Paradigm	Conversational Programming: control emerges from the flow of dialogue	Role engineering: control is pre-conditioned by strict prompts and roles	Formal Specification: the control is encoded in a DSL within a managed runtime	Minimal Toolkit: control is defined by custom composition of modules
Pattern Architecture Agent	Conversational agent, whose behaviour is programmed and orchestrated through the dialogue flow	Role-driven agent, whose behaviour is pre-conditioned by an initial 'script' ('Inception Prompting')	Agent as a formally specified process whose logic is encoded in a DSL within a structured operational environment	Agent as a minimal assembly of functional modules, designed as a 'meta-framework' for experimentation
Approach to Agent Deliberation	Deliberation as dialogue: conversation history guides reasoning	Deliberation as acting: the 'Inception Prompt' binds reasoning to a role	Deliberation as a process: the DSL orchestrates LLM calls within an explicit logic	Deliberation as a strategy: the ReAct/Reflection cycle can be implemented at module level
Transparency of the Resolution (vs. BDI)	Opaque (internal reasoning within the LLM); traceability is an external mechanism (e.g. logging) for reconstructing the process	Opaque; the 'prompt' of the role guides the reasoning, but the internal process cannot be inspected. External tracing	Partially transparent via DSL, but the LLM decision-making core remains opaque. Process-level tracing	Opaque; modularity allows inspection of component inputs/outputs, but not the internal reasoning of the LLM
II. IMPLEMENTATION OF AOSE PRINCIPLES (A&A MODEL)				
Environmental Engineering	Implicit and mediated by conversation	Implicit and mediated by shared roles and tasks	Explicit and based on defined resources (Readers, Functions)	Implementation-dependent (Actions could act as artifacts)
Implementation of Coordination Artifacts	Implicit (e.g. GroupChatManager)	Implicit (e.g. Workforce task channel)	Explicit (e.g. Readers and Functions)	Not supplied out-of-the-box, to be built

Continued on next page

Table 4.5 – continued from previous page

Comparison Criterion	AutoGen	CAMEL	Eclipse LMOS Arc	AgentLite
Organisational Formalism	Implicit organisation (emerging from dialogue) or programmatic organisation (e.g. Manager); not based on formal specifications	Organisation based on textual roles and hierarchies (Workforce); paradigm different from formal specifications	To check	Flexible/Non-prescriptive; typically programmatic (e.g. Manager Agent)
III. SYSTEM ENGINEERING				
Multi-Agent Interaction Support	Conversational patterns (GroupChat, Debate, Handoffs) and graph orchestration (GraphFlow)	Role-based dialogue (Society), hierarchical architecture (Workforce), large-scale simulations (OASIS)	DSL constructs (callAgent, AgentChain) supported by LMOS platform (discovery, routing)	Orchestration through 'manager agents' in hierarchical structures; less prescriptive communication
Debugging-Observability	High (advanced logging, native integration with OpenTelemetry)	High (through integration with external services such as AgentOps for tracing and replay)	Very high (Micrometer tracing, custom events, Arc View graphic UI)	Good (intrinsic to clarity and separation of components)
Modularity-Reusability	Very high (agents and tools as reusable and modular units)	Very high (modular architecture with specialised agents and modules, Toolkits)	Very high (thanks to DSL, bean spring, reusable functions, remote agents)	Very high (inherently modular design of PromptGen modules, Actions, etc.)

4.8 Towards an Engineered Synthesis: Integration Patterns

The analysis conducted so far has highlighted a tension between the wealth of knowledge possessed by generative agents and the less explicit structure of their frameworks [SYNG23]. As the previous chapters have demonstrated the specific architectural trade-offs of this approach, the solution does not lie in rejecting one paradigm in favour of the other, but in pursuing an engineered synthesis capable of integrating the strengths of both paradigms. Below is an outline of three potential architectural patterns that can guide this integration.

The LLM as a Deliberative Component in a BDI Agent

In this pattern, the LLM is not the agent, but becomes a specialised component within a traditional BDI architecture [Woo09]. The BDI interpreter maintains control of the practical reasoning cycle (Belief-Desire-Intention). However, instead of relying solely on a static plan library, the agent can invoke the LLM at specific moments:

- **Dynamic Plan Generation**: when faced with an unexpected event for which there is no predefined plan, the agent can ask the LLM to generate a new plan in natural language, which is then parsed and added to the plan library;
- **Sophisticated Option Valuation**: when filtering desires to form intentions, the agent can use the LLM to evaluate complex options, leveraging its vast knowledge of the world for richer deliberation. This pattern maintains the formal guarantees of the BDI cycle, enriching it with the creative flexibility of the LLM.

AOSE artefacts as "Smart Tools" for Generative Agents

Instead of having a generative agent (e.g. based on ReAct [YZY+22]) interact directly with a raw API, the interaction is mediated by a CArtAgO artefact [WOO07][ORV08][RPVO09][ORV+04]. The artefact acts as a "smart shell" around the tool, presenting the agent with a structured and secure interface. The advantages are immediate:

- Interaction with an artefact, based on the invocation of formal operations, would eliminate the reliability challenges of docstring-based JSON call generation, which is typical of current generative agents;
- The artefact manages the interaction status and can generate structured events (e.g. fileClosed, auctionEnded), providing the agent with clear and unambiguous perceptions;
- The complex coordination logic (e.g. a resource reservation mechanism) is encapsulated in the artefact, lightening the cognitive load on the LLM.

Moise for the Governance of Generative Agent Societies

To introduce a formal governance model in multi-agent systems such as AutoGen, the organisational structure is not defined programmatically and implicitly, but through an explicit and external Moise specification [HSB07][HSB02]. This specification defines roles (e.g. Programmer, Tester), rules (OBLIGATION to perform tests before committing) and social patterns (Feature Development). A "guardian agent" or the environment itself [WOO07] can therefore monitor interactions between generative agents to ensure their compliance with organisational specifications, introducing a level of formal, verifiable and dynamically reconfigurable governance that would complement existing programmatic governance mechanisms.

However, this summary should not be one-sided. While AOSE provides architectural patterns for structured governance, generative agents offer the intelligent connective tissue to overcome the historical limitations of AOSE itself. The transformative potential of LLMs can revolutionise agent-oriented software engineering practice in ways that have only been hypothesised until now:

- **AOSE Modelling Automation**
  The creation of formal specifications (Moise, artefact interfaces) is a knowledge-intensive activity that represents a barrier to the adoption of AOSE. An advanced LLM could act as an "assistant AOSE engineer", generating draft Moise specifications [HSB07][HSB02] or interfaces for CArtAgO artefacts [WOO07][ORV08][RPVO09][ORV$^+$04] from a description of requirements in natural language. This would drastically reduce adoption costs and make formal modelling accessible even to non-specialists;
- **BDI agents with enhanced and adaptive reasoning**
  The plan library of a BDI agent [Woo09] is traditionally static and pre-programmed, limiting its adaptability. By integrating an LLM as a deliberative component, a BDI agent could not only generate plans dynamically (as already hypothesised), but also perform much more sophisticated belief revision. Faced with contradictory perceptions, the agent could use the LLM to reason about the possible causes of the discrepancy and update its model of the world in a more flexible and contextual way, overcoming the limitations of classical formal logic;
- **Intelligent and Self-Explanatory Artefacts**
  AOSE artefacts [WOO07], while robust and designed as active entities for mediating interactions, typically operate at a procedural and non-semantic level. We could imagine 'intelligent artefacts' whose internal behaviour is governed by a specialised LLM. For example, a 'coordination whiteboard' artefact could store information, but also use an LLM to summarise discussions, identify conflicts between agents' plans, and autonomously suggest resolution strategies.

This bidirectional vision outlines a research perspective where the goal is an architecture in which the formal structure of AOSE and the semantic fluidity of LLMs are in productive symbiosis, favouring the development of autonomous systems that are simultaneously intelligent, robust, and governable.

Chapter 5

Comparative Analysis between A&A Patterns and Generative Framework Architectures

The preceding chapters have established a theoretical framework for analysing generative agent frameworks, introducing the principles of AOSE as a critical lens. It has been argued that, despite their rapid evolution, these frameworks are developing solutions to fundamental problems of coordination, mediation and abstraction already formalised by AOSE. This represents a functional convergence: the need to solve the same engineering challenges.
This chapter shifts the analysis from the theoretical to the empirical level, with the aim of conducting a rigorous comparative analysis of this functional convergence. It will explore how different architectural paradigms - the environment-centric approach of AOSE/CArtAgO and the agent-centric or service-oriented approaches of modern frameworks - address similar challenges.
The analysis will proceed by dissecting three specific implementations that functionally reflect the fundamental architectural patterns of the A&A meta-model. Three distinct patterns will be examined: the coordination artefact, the mediation artefact, and the functional artefact. For each pattern, an implementation taken from a modern framework will be compared with its conceptual counterpart in CArtAgO, highlighting functional similarities and, above all, architectural differences.
The central thesis of this chapter is that, while modern frameworks converge towards the need for environmental abstractions, their implementation reveals a clear paradigmatic divergence. By choosing to implement these abstractions as specialised agents (an agent-centric approach) or as decoupled services (a service-oriented approach), rather than as first-class environmental entities (the environment-centric approach of AOSE), they adopt alternative design choices that introduce a different set of engineering trade-offs. This analysis will compare these different architectural choices and their respective trade-offs in terms of robustness, modularity, and observability, using established models as a benchmark for the environment-centric approach.

5.1 The Coordination Artifact Pattern: AutoGen's BaseGroupChatManager

AutoGen's analysis of BaseGroupChatManager[1] provides a first case study on AOSE principles. Although defined as an agent (see SequentialRoutedAgent), its primary functional role is that of an interaction protocol coordinator, rather than an autonomous participant with its own objectives. Its purpose is to govern the 'laws of social physics' of a group conversation — a role that in AOSE is typically delegated to an environmental artefact.
To analyse this implementation in detail, we will compare it with the established architectural pattern of the A&A coordination artefact. We will use a CArtAgO implementation, referred to here as MeetingRoomArtifact[2], as the canonical reference for this specific pattern, which faithfully embodies the AOSE principles - observable state, public operations and signals. A direct comparison between BaseGroupChatManager and this canonical reference reveals functional equivalence,

[1] The complete source code for the implementation is available at BaseGroupChatManager.py
[2] The complete source code for the CArtAgO reference analysis and implementation is available at MeetingRoomArtifact.java

masked by profound architectural divergence.
The signature of the class itself reveals the fusion of roles and its nature as a 'template' for an environmental protocol:

```
class BaseGroupChatManager(SequentialRoutedAgent, ABC):
    ...

class SequentialRoutedAgent(RoutedAgent):
    ...
    def __init__(
        self,
        description: str,
        sequential_message_types: Sequence[type[Any]])
    -> None:
        super().__init__(description=description)
        self._fifo_lock = FIFOLock()
        self._sequential_message_types = sequential_message_types
```

The inheritance of BaseGroupChatManager from SequentialRoutedAgent imposes a mechanical sorting role (a lock) on it, while its ABC nature defines it as a protocol template (imposing select_speaker and reset), not as a complete deliberative entity: its function is not to decide, but to execute and enforce a protocol.

5.1.1 Encapsulation of the Interaction State

A CArtAgO artefact, by its very nature, reifies the state of an interaction, making it an observable and shareable property of the environment. The BaseGroupChatManager achieves the same goal, but by encapsulating this state internally: its instance variables correspond directly to the observable properties of an artefact.

```
#...
# Equivalent to defineObsProperty()
self._participant_names = participant_names     # participants list
self._message_thread: List =     # conversation history
self._current_turn = 0     # turn counter
self._active_speakers: List[str] =     # current active speakers
#...
```

The crucial difference lies in the exposure mechanism: in CArtAgO, the state is explicitly published to the environment via defineObsProperty("current_turn", currentTurn), making it a public fact that can be inspected by any authorised agent; in AutoGen, the state is an internal implementation detail of an agent, and other participants infer the state only through the messages they receive. For example, an agent does not observe a current_speaker state property; instead, it receives a GroupChatRequestPublish message and infers from this that it is its turn to speak. Therefore, the state is communicated rather than observed.

5.1.2 Operations Governed as Interaction Protocol

Both constructs serve to define a formal set of rules that determine valid behaviour within the interaction. In CArtAgO, this protocol is defined by a set of methods annotated with @OPERATION; in AutoGen, methods decorated with @rpc and @event define the public interface of the BaseGroupChatManager.

```
@event
async def handle_agent_response(
    self,
    message: GroupChatAgentResponse,
    ctx: MessageContext) -> None:
        #...
        # Management of active speakers
        self._active_speakers.remove(message.agent_name)
        if len(self._active_speakers) > 0:
            return  # Still waiting for other speakers
        #...
```

The internal logic of handle_agent_response further demonstrates its environmental role. The operation self._active_speakers.remove(message.agent_name) is not a deliberative act, but a direct manipulation of the coordination state.

5.1.3 Logics of Rotation and Signalling as Environmental Laws

Turn management is an example of a coordination rule. The BaseGroupChatManager implements this logic in the _transition_to_next_speakers method: it does not simply select the next speaker, but actively publishes a request on their specific topic, an action that is functionally identical to sending a signal in CArtAgO.

```
async def _transition_to_next_speakers(
    self,
    cancellation_token: CancellationToken) -> None:
        # ...
        # Activation of selected speakers
        # (signal("permission_to_speak", speaker) in CArtAgO)
        for speaker_name in speaker_names:
            speaker_topic_type =
                self._participant_name_to_topic_type[speaker_name]
            await self.publish_message(
                GroupChatRequestPublish(),
                topic_id=DefaultTopicId(type=speaker_topic_type),
                cancellation_token=cancellation_token,
            )
            self._active_speakers.append(speaker_name)
```

The call await self.publish_message(GroupChatRequestPublish(),...) is the functional equivalent of CArtAgO's signal("permission_to_speak", selectedSpeaker).

5.1.4 Life Cycle Management: Termination and Reset

Functional convergence also extends to interaction lifecycle management.

```
async def _apply_termination_condition(
    self,
    delta: Sequence,
    increment_turn_count: bool = False
) -> bool:
    """Termination logic"""
    # Custom condition check
    if self._termination_condition is not None:
        stop_message = await self._termination_condition(delta)
        if stop_message is not None:
            # Status reset
            await self._termination_condition.reset()
            self._current_turn = 0
            # Termination notification
            await self._signal_termination(stop_message)
            return True

    # Increase shift counter
    if increment_turn_count:
        self._current_turn += 1

    # Checking the maximum number of turns
    if self._max_turns is not None and self._current_turn >= self._max_turns:
        stop_message = StopMessage(
            content=f"Maximum number of turns {self._max_turns} reached.",
            source=self._name,
        )
        # Reset and signal
        if self._termination_condition is not None:
```

```
            await self._termination_condition.reset()
        self._current_turn = 0
        await self._signal_termination(stop_message)
        return True

    return False
```

The BaseGroupChatManager implements explicit termination logic in the _apply_termination_condition method, which checks both a custom condition and a maximum number of turns. This mechanism is an implementation of the logic present in the MeetingRoomArtifact, where checkTermination-Condition and the maxTurns parameter perform the exact same function.

```
@rpc
async def handle_reset(
    self,
    message: GroupChatReset,
    ctx: MessageContext) -> None:
        """Reset operation"""
        await self.reset()

@abstractmethod
async def reset(self) -> None:
    """Abstract reset operation - must be implemented by subclasses"""
    ...
```

Similarly, the reset operation is present in both constructs. In AutoGen, the @rpc async def handle_reset(...) method acts as a public endpoint that invokes the reset() logic, returning the interaction to its initial state. This is the functional counterpart of CArtAgO's @OPERATION public void resetMeeting().

This architectural choice introduces an engineering trade-off. On the one hand, by coupling the coordination logic to the agentive communication model (inheriting from RoutedAgent), the framework ensures that the coordinator natively 'speaks' the agent protocol. On the other hand, this solution offers a different type of modularity: the coordination logic is coupled to the agent model (RoutedAgent), promoting reusability at the agent level rather than at the environment level - as in the case of the CArtAgO artefact. This can make it more complex to reuse in contexts that do not adopt the same agent model.
The following table summarises the direct comparison, highlighting functional equivalence and implementation divergence.

Table 5.1: Comparative analysis between MeetingRoomArtifact (CArtAgO) and BaseGroupChatManager (AutoGen)

AOSE/CArtAgO concept	**CArtAgO (MeetingRoomArtifact)**	**AutoGen (BaseGroupChatManager)**	**Notes**
Status Management	private List<String> messageThread; private int currentTurn;	self._message_thread: List[...] self._current_turn=0	Functionally identical state, but private to an agent and not observable in the environment
Definition of Operations	@OPERATION public void handleAgentResponse(...)	@event async def handle_agent_response(...)	Both define a public interface. @OPERATION is for an environmental entity, @event/@rpc for an agent.

Continued on next page

Table 5.1 – continued from previous page

AOSE/CArtAgO concept	CArtAgO (MeetingRoomArtifact)	AutoGen (BaseGroupChatManager)	Notes
Internal Logic	public void selectNextSpeaker() ...	async def _transition_to_next_speakers(...) ...	The turn logic is implemented in both, but the location of execution differs (environment vs. agent)
Signalling/Output	signal ("permission_to_speak", ...)	await self.publish_message(...)	CArtAgO uses synchronous signals linked to observable properties. AutoGen uses an asynchronous publish/subscribe mechanism.

5.2 The Mediation Artifact Pattern: AutoGen's UserProxyAgent

AutoGen's UserProxyAgent provides a second case study of convergence towards A&A patterns, specifically that of the mediation artefact. The primary role of this agent is not autonomous deliberation, but the abstraction of an external resource - a human user - acting as a translator between the world of human I/O and the world of the multi-agent system.

5.2.1 Reactive Logic and Lack of Deliberation

Analysis of the main operating cycle, the on_messages_stream method, shows primarily reactive behaviour, designed to handle the flow of I/O to and from the user.

```
async def on_messages_stream(
    self,
    messages: Sequence,
    cancellation_token: CancellationToken
) -> AsyncGenerator:
try:
    # (1) Check incoming messages (for handoff)
    handoff = self._get_latest_handoff(messages)
    #...
    # (2) Emits an input request event
    input_requested_event = UserInputRequestedEvent(...)
    yield input_requested_event

    # (3) Waiting for external input
    user_input = await self._get_input(prompt, cancellation_token)

    # (4) Wrap the response in a system format
    if handoff:
        yield Response(chat_message=HandoffMessage(...))
    else:
        yield Response(chat_message=TextMessage(...))
#...
```

The control flow is a perception-action cycle focused on I/O. Although the main deliberation (the response) is delegated to the user, the agent itself possesses non-trivial internal logic. As evident from the above source code, the agent performs active protocol validation, for example by handling handoff logic and raising errors if messages are not addressed correctly (see the next code snippet). This configures it as a specialised mediation agent that encapsulates the logic of interaction, rather than as a purely passive entity.

5.2.2 Error Handling and System Protection

A classic task of a mediation artefact is to 'protect' the multi-agent system from unexpected behaviour and failures of the external resource. The UserProxyAgent performs this role through its error handling, which translates external exceptions into controlled system errors.

```
async def _get_input(
    self,
    prompt: str,
    cancellation_token: Optional) -> str:
try:
    # ...I/O logic...
except asyncio.CancelledError:
    # Cancellation management
    raise
except Exception as e:
    # Conversion of external errors to the system format
    raise RuntimeError(f"Failed to get user input: {str(e)}") from e
```

The translation of a generic Exception into a system RuntimeError is an act of mediation: it prevents an unpredictable failure of the external resource (e.g. an I/O error) from propagating in an uncontrolled manner, ensuring that the system remains in a stable state.

5.2.3 Impedance Mismatch Management

Interaction with external resources often introduces architectural "impedance mismatch". The UserProxyAgent's _get_input method is a mechanism for resolving this issue, handling both synchronous and asynchronous inputs.

```
async def _get_input(
    self,
    prompt: str,
    cancellation_token: Optional) -> str:
    #...
    if self._is_async:
        # async
        return await async_func(prompt, cancellation_token)
    else:
        # sync
        loop = asyncio.get_event_loop()
        return await loop.run_in_executor(None, sync_func, prompt)
    #...
```

The use of loop.run_in_executor to execute a synchronous function in a separate thread is a critical engineering feature that prevents the main event loop from being blocked. In the AOSE paradigm, such complexity would be entirely encapsulated within the body of the artefact.

However, the signature of the class, class UserProxyAgent(BaseChatAgent,...) , reveals a design choice specific to AutoGen's agent-centric paradigm. Its main functions are I/O management, protocol translation, and resource abstraction — all responsibilities that the AOSE/A&A paradigm would treat as environmental. Framing this component as an 'agent' maintains the internal consistency of the AutoGen model (where everything that acts is an agent), but represents a design choice that differs from the clear separation of responsibilities that the A&A model formally implements.

5.3 The Functional Artifact Pattern: Eclipse LMOS Arc Functions

The analysis of the Functions mechanism in Eclipse LMOS Arc presents a case study of a structurally more explicit convergence towards the AOSE pattern of the functional artefact. Although still operating within an implicit paradigm, this framework is closer to AOSE design principles, particularly with regard to separation of responsibilities and decoupling.

5.3.1 Explicit Separation between Interface and Implementation

LMOS Arc formally separates the definition of a tool from its execution logic, mirroring the AOSE distinction between an artefact's user interface and its internal functioning. The productsearch.functions.kts file defines a declarative and formal interface for the productsearch function, specifying its name, description, and parameters. This is the 'contract' that the agent will use to interact with the functionality.

```
function(
    name = "productsearch",
    description = "Search products based on the specifications",
    params = types(
        string(
            name = "query",
            description = "A query with technical descriptions..."
        )
    ),
) { (query) ->... }
```

The actual business logic is contained in a separate file, ProductSearch.kt, within a class annotated with @Component. This two-part structure is a direct parallel to a CArtAgO artefact, where the artefact code defines the @OPERATION signatures (the interface) and the method bodies contain the implementation.

5.3.2 Decoupling through Dependency Injection

The mechanism that achieves true decoupling is the get<ProductSearch>() call. The function wrapper in productsearch.functions.kts does not directly instantiate the ProductSearch class; instead, it requests an instance from the system runtime:

```
// Injection dependency
val productSearch = get<ProductSearch>()  // Access to the backend service

// Operation delegation
val contextResult = query?.let {
    productSearch.searchProduct(it, "")  // Call to the service method
}
```

This represents a Dependency Injection (DI) pattern. In the AOSE paradigm, the environment itself acts as a DI container: agents look up artefacts by name without needing to know their implementation class. LMOS Arc's use of DI is an engineering choice that achieves the same level of decoupling.
This architecture represents a point of greater structural alignment: LMOS Arc successfully reinvents the look and feel of a functional artefact, offering a decoupled, runtime-managed service.
It is important, however, to distinguish between the specific example and the potential of the pattern. The example analysed (ProductSearch.kt) is stateless in nature: it acts as a simple gateway for an external API. However, this stateless nature is not an intrinsic limitation of the architectural pattern, as the Dependency Injection mechanism (with @Component) would technically allow stateful services (e.g. singletons) to be managed.
Despite this potential, the emphasis of the LMOS Function pattern seems to be on exposing stateless business logic, and therefore deviates from CArtAgO's goal of providing complete environmental entities with observable state, their own lifecycle, and the ability to act proactively.

5.4 Summary: Comparative Analysis of Architectural Trade-Offs

The empirical analysis conducted in this chapter provides concrete evidence, at the code level, of the project's central thesis. Examination of BaseGroupChatManager, UserProxyAgent, and LMOS Arc Functions demonstrates that modern frameworks for generative agents are independently and implicitly developing solutions for coordination, mediation, and functional extension.
The divergence identified does not lie in the functionality achieved, but in the location of implementation. By implementing these environmental responsibilities as specialised agents - AutoGen,

an agent-centric paradigm - or as service wrappers - LMOS Arc, a service-oriented paradigm -, these frameworks make an architectural choice that differs from AOSE.
The fundamental engineering insight behind AOSE is that the environment itself should be a first-class computational entity. Modern frameworks, on the other hand, favour a 'lightweight' environment - often a message bus - and encapsulate coordination and mediation intelligence within other entities - agents or services.
The engineering trade-offs of this alternative approach are tangible and manifest themselves in several design considerations when compared to the A&A model:

- **Observability (Communicated vs. Observed State)**
  In an agent-centric model, the state is private and communicated via messages: external inspection requires monitoring this message flow. In an environment-centric model (CArtAgO), the state is public and observed (defineObsProperty), allowing direct inspection of the artefact's state;

- **Modularity and Coupling (Agent vs. Environment)**
  The agent-centric approach couples coordination logic with the agent model (e.g. RoutedAgent), promoting reusability at the agent level. The environment-centric approach decouples logic from the agent, promoting reusability at the environment level;

- **Governance Model (Local vs. Global)**
  Agent-centric governance is local: the "laws" are encapsulated in the manager and apply to the agents interacting with it. AOSE's environment-centric governance is global: the laws embedded in the environment can be made applicable to all entities inhabiting that environment.

In conclusion, the emergence of these patterns validates the fundamental principles of AOSE - demonstrating 'functional convergence' -, while the different implementation strategies - demonstrating 'paradigmatic divergence' - highlight the existence of multiple valid architectural approaches to solving the same problems. This opens up a prospect for engineering synthesis: the possibility of future hybrid architectures that can combine the generative power and flexibility of modern frameworks with the robustness and observability patterns offered by treating the environment as a first-class entity, drawing on the best of both paradigms.

Chapter 6

Empirical Analysis of Organisational Divergence: A Dimensional Comparison with Moise

This chapter presents an empirical validation of the central thesis of this paper, introduced in Chapter 1. The previous analysis argued that, despite rapid evolution and "functional convergence" towards the classic problems of AOSE, modern generative agent frameworks exhibit a clear "paradigmatic divergence". The fundamental engineering problem remains how to coordinate deliberative components that are inherently stochastic and not immediately inspectable, in order to achieve structured collective behaviour.
The paradigmatic divergence lies in how the multi-agent organisation is realised. While the AOSE paradigm has focused on developing explicit and declarative organisational models to ensure robustness and verifiability, emerging generative frameworks adopt alternative approaches, primarily implicit and programmatic, where the social structure is realised through procedural logic and message passing.
To empirically examine this divergence, this chapter performs a dimensional analysis: the Moise model, introduced in Chapter 2, is used as an analytical benchmark to compare the nature of organisational specifications. The analysis will map the source code[1] of two representative frameworks with respect to Moise's three orthogonal dimensions:

- **Structural Specification (SS)**
  Defines the static social architecture and answers the question: "Who is part of the organisation and how are they connected?". It includes the definition of Roles (behavioural abstractions), Groups (aggregations of roles) and Links (social relationships of communication, authority or knowledge);

- **Functional Specification (FS)**
  Defines collective objectives and their breakdown. Answers the question: "What does the organisation do?". It Includes Schemes (hierarchical global plans) and Missions (specific objectives assigned to roles);

- **Deontic Specification (DS)**
  Defines the normative rules that connect structure (SS) to function (FS). Answers the question: "What are the rules of the game?" and uses deontic concepts such as Obligations (what a role must do) and Permissions (what a role can do).

By applying this analytical lens to the source code of AutoGen and CAMEL, we aim to empirically demonstrate how organisation in these systems is programmed and implicit, identifying dimensional differences that illustrate paradigmatic divergence.

[1] The complete source code for the implementation is available at GitHub project

6.1 Case Study 1: Flat and Programmatic Organisation (AutoGen BaseGroupChat)

The first case study analyses AutoGen's BaseGroupChat class. This implementation is representative of a non-hierarchical approach to organisation, in which collaboration is primarily conceived and managed as a conversational protocol. As discussed in Chapter 5, governance is delegated to a Manager agent, whose role is procedural.

6.1.1 Structural Dimension: A Static and Predefined Architecture

Moise's Structural Specification (SS) is present, but its nature is entirely programmatic and static. The Roles (Moise) find a direct correspondence in the ChatAgent instances passed as a list in the participants argument to the class constructor (__init__). The Group (Moise) is the BaseGroupChat instance itself, identified by a unique _team_id.

```
class BaseGroupChat(Team, ABC, ComponentBase):

def __init__(
    self,
    participants: List[ChatAgent],     # Role definitions
    group_chat_manager_name: str,     # Coordinator role
    #...
):
    #...
    # Team identity
    self._team_id = str(uuid.uuid4())
```

The constructor analysis reveals the static and programmatic nature of this structure; the composition of the group is fixed at instantiation time. The code performs immediate validation (e.g. if len(participants) == 0: raise ValueError(...)) and populates immutable internal lists (such as self._participant_names).

```
# Validation of the organisational structure
if len(participants) == 0:
    raise ValueError("At least one participant is required.")
if len(participants)!=
    len(set(participant.name for participant in participants)):
    raise ValueError("The participant names must be unique.")

# Structural specification
self._participants = participants
#...
self._participant_names: List[str] =
    [participant.name for participant in participants]
```

Links (Moise), which define social relationships such as authority or knowledge, have an indirect correspondence. They are implemented as communication topics (e.g. self._group_topic_type and self._participant_topic_types).

```
# Communication topology similar to Moise links
self._group_topic_type = f"group_topic_{self._team_id}"     # Broadcast channel
self._group_chat_manager_topic_type =
    f"{group_chat_manager_name}_{self._team_id}"     # Manager channel
self._participant_topic_types: List[str] = [     # Individual channels
    f"{participant.name}_{self._team_id}" for participant in participants
]
```

This is a difference in engineering approach: AutoGen defines a messaging topology - a pub/sub architecture - and uses it as a substitute for the social organisational structure. There is no explicit concept of authority or knowledge links as specified in Moise; these relationships are handled implicitly through the message topology and the Manager logic.

6.1.2 Functional Dimension: Unified Task Management

The Functional Specification (FS) is present, but implemented in a programmatic rather than hierarchical form.
The Mission (Moise) corresponds to the task argument (whether str or BaseChatMessage) passed to the run_stream method. This task is processed and encapsulated in a GroupChatStart message, which effectively starts the execution of the collective objective.

```
async def run_stream(
    self,
    *,
    task: str | BaseChatMessage | Sequence | None = None,
    cancellation_token: CancellationToken | None = None,
) -> AsyncGenerator[...]:
    """␣Organisational␣implementation␣"""

    # Task processing, equivalent to mission
    messages: List | None = None
    if task is None:
        pass
    elif isinstance(task, str):
        messages = [TextMessage(content=task, source="user")]
    # ... (processing other types of tasks)

    try:
        # Mission launch
        await self._runtime.send_message(
            GroupChatStart(messages=messages),
            recipient=
                AgentId(
                    type=self._group_chat_manager_topic_type,
                    key=self._team_id
                ),
            cancellation_token=cancellation_token,
        )
        # ...
```

The key architectural difference in this dimension is the absence of explicit Schemas (Moise): no formal or structural representation is used for the hierarchical breakdown of the objective, and the task provided is managed in a unitary manner. The breakdown of the objective is not a pre-specified structural artefact, but is managed as an emergent process that arises from the dynamics of the conversation.
As analysed in Chapter 5, this decomposition is implicitly managed by the procedural governance of GroupChatManager, which orchestrates speaking turns, but the system itself does not have an explicit representation of sub-objectives or a plan.

6.1.3 Deontic Dimension: Governance as Procedural Logic

The analysis of the Deontic Dimension (DS) reveals the most marked paradigmatic difference. Governance is implemented differently, i.e. not as a separate declarative construct (as in Moise), but as an integral part of procedural logic.
Analysing the code, we can see that the concepts of Obligations or Permissions (Moise) are not defined as explicit, separate constructs. Instead, the 'rules' governing interaction are implemented as programming constraints and execution logic. The clearest examples are the arguments termination_condition and max_turns:

```
def __init__(
    self,
    participants: List[ChatAgent],
    # ...
    termination_condition: TerminationCondition | None = None, # Goal completion
    max_turns: int | None = None,     # Constraints
    # ...
```

```
):
    # ...
    # Coordination configuration
    # ...
    self._termination_condition = termination_condition # Objective termination
    self._max_turns = max_turns     # Turns limit
```

These constraints highlight the paradigmatic divergence: the termination_condition argument is not a consultable deontic norm, but a procedural control. The system bases its governance on the evaluation of programmatically defined execution conditions, rather than on the consultation of an explicit normative state - as in Moise.

As discussed in Chapter 5, the main 'law' governing the system is the procedural governance logic imposed by the GroupChatManager. The turn-taking logic (e.g. round-robin or LLM-based selection) is the only 'rule' that agents follow, and it is not an external, consultable specification but is hard-coded into the Manager's logic, making governance purely procedural. Thus, the rule is not absent but is implicit in the Manager's code.

6.2 Case Study 2: Structural Convergence in Hierarchical Organisation (CAMEL Workforce)

The second case study analyses CAMEL's Workforce class, an implementation that represents an organisational architecture with a more explicit hierarchical structure. As the analysis will show, Workforce exhibits a much stronger 'functional convergence' towards Moise's structural and functional principles, while maintaining the same difference in approach in the deontic dimension.

6.2.1 Structural Dimension: The Emergence of Explicit Roles and Hierarchies

The Structural Specification (SS) in CAMEL Workforce is implemented with greater structural detail than that of AutoGen. Firstly, it defines explicit, specialised Roles with distinct responsibilities. Analysis of the ___init___ method shows the instantiation of agents with predefined roles: self.coordinator_agent (identified as 'Workforce Manager') and self.task_agent (identified as 'Task Planner').

```
class Workforce(BaseNode):
# ...
def __init__(
    self,
    # ...
    coordinator_agent_kwargs: Optional = None, # Coordinator Configuration
    task_agent_kwargs: Optional = None, # Planner Configuration
    # ...
) -> None:
    # ...
    # Coordinating agent
    coord_agent_sys_msg = BaseMessage.make_assistant_message(
        role_name="Workforce␣Manager",
        content="You␣are␣coordinating␣a␣group␣of␣workers...",
    )
    self.coordinator_agent = ChatAgent(
        coord_agent_sys_msg,
        **(coordinator_agent_kwargs or {}),
    )

    # Planning agent
    task_sys_msg = BaseMessage.make_assistant_message(
        role_name="Task␣Planner",
        content="You␣are␣going␣to␣compose␣and␣decompose␣tasks...",
    )
    # ...
```

```
        self.task_agent = ChatAgent(task_sys_msg, **_task_agent_kwargs)
```

This represents a convergence towards AOSE, as CAMEL architecturally separates the competence of coordination from that of planning, a key principle of organisational design.
Secondly, CAMEL explicitly implements hierarchy (Moise Groups and Subgroups). This is evident in the attribute self._children: [Optional], designed to contain subordinate workers or entire sub-organisations. The method def add_workforce(self, workforce: Workforce) serves as an explicit programmatic interface for building a hierarchy of nested groups.

```
def __init__(
    self,
    # ...
    children: Optional] = None,     # Hierarchical structure
    # ...
) -> None:
    super().__init__(description)
    self._children = children or []     # Sub-organizations/workers
    # ...

def add_workforce(self, workforce: Workforce) -> Workforce:
    """␣Add␣sub-organisation␣(add␣a␣Moise␣sub-group)␣"""
    self._children.append(workforce)
    return self
```

Although CAMEL's SS strongly converges with Moise's concepts, it remains programmatic: the organisation is built through method calls (e.g. add_workforce) and passing configurations to the constructor, not by loading an external declarative specification.

6.2.2 Functional Dimension: Breaking Down the Goal as a First-Class Function

The Functional Specification (FS) in CAMEL is a first-class construct and represents the point of greatest convergence.
Unlike AutoGen's uniformly managed TASK: STR, CAMEL uses explicit classes to represent the function: the organisation receives a TASK: TASK and the Task Planner Agent is responsible for populating its SUBTASKS: LIST attribute. The FS is therefore represented as a data structure (a task tree), not as a simple string. This allows the system to track, delegate, assign, and compose the results of sub-objectives explicitly (as seen in task.result = ...).

```
async def process_task_async(
    self,
    task: Task,
    interactive: bool = False) -> Task:
    # ...
    # Task decomposition (task_planner goal)
    subtasks = self._decompose_task(task)
    # ...
    if subtasks:
        # ...
        self._pending_tasks.extendleft(reversed(subtasks))
    else:
        self._pending_tasks.append(task)
    # ...
    # Composition of results
    if subtasks:
        task.result = "\n\n".join(
            f"——␣Subtask␣{sub.id}␣Result␣——\n{sub.result}"
            for sub in task.subtasks
            if sub.result
        )
        # ...
    return task
```

The _decompose_task method is the programmatic implementation of a Moise Scheme. It takes a global objective (the task) and, delegating to the appropriate Role (the Task Planner Agent), breaks it down into sub-objectives (subtasks).

```
def _decompose_task(self, task: Task) -> List:
    """␣Task␣decomposition␣(possible␣decompose_and_plan␣goal)"""
    decompose_prompt = WF_TASK_DECOMPOSE_PROMPT.format(
        content=task.content,
        child_nodes_info=self._get_child_nodes_info(),
        additional_info=task.additional_info,
    )
    self.task_agent.reset()
    subtasks = task.decompose(self.task_agent, decompose_prompt)
    task.subtasks = subtasks
    for subtask in subtasks:
        subtask.parent = task
    return subtasks
```

These subtasks become Missions that the Coordinator Agent will assign to workers. This explicitly links the SS (the role) to the FS (the decomposition function), adhering to Moise's principles.

6.2.3 Deontic Dimension: Norm as Programmatic "Capability"

Despite convergence in structural and functional dimensions, analysis of the deontic dimension in CAMEL confirms the thesis of paradigmatic divergence: the difference in the deontic approach persists.
The most clear example is the Permission (Moise) to dynamically create new workers to adapt the organisation. In the Workforce code, this capability is implemented as a Python method:

```
def _create_worker_node_for_task(self, task: Task) -> Worker:
    """ Dynamic worker creation (permission "create_new_workers" in Moise) """
    prompt = CREATE_NODE_PROMPT.format(
        content=task.content,
        child_nodes_info=self._get_child_nodes_info(),
        additional_info=task.additional_info,
    )
    response = self.coordinator_agent.step(prompt, response_format=WorkerConf)

    if response.msg is None or response.msg.content is None:
        # Fallback strategy
        new_node_conf = WorkerConf(
            description=f"Fallback worker for task: {task.content[:50]}...",
            role="General Assistant",
            sys_msg="You are a general assistant that
                        can help with various tasks.",
        )
    else:
        result_dict = json.loads(response.msg.content)
        new_node_conf = WorkerConf(**result_dict)

    # Create new agent
    new_agent = self._create_new_agent(
        new_node_conf.role,
        new_node_conf.sys_msg
    )

    # Worker node instantiation
    new_node = SingleAgentWorker(
        description=new_node_conf.description,
        worker=new_agent,
        pool_max_size=10,
    )
    #...
```

```
        # Hierarchical integration
        self._children.append(new_node)
        #...
        return new_node
```

The rule is implemented programmatically as part of the procedure. The Coordinator Agent does not consult an external specification to determine whether it has permission to create a new worker in a given situation. This is an example of an agent-centric and procedural norm, since the 'permission' is encoded as a 'capability' (a method) of the agent. In fact, the very existence of the _create_worker_node_for_task method is the permission, and the agent executes the defined procedure rather than consulting an external specification. Its 'deliberation' – delegated to a prompt – concerns how to configure the worker, not whether the action is permitted - since the existence of the method implies this.
This implementation confirms that, even in an advanced framework, governance is a procedural design choice, based on capabilities defined in the code, rather than on a specific declarative (deontic) specification.

6.3 Critical Summary: The Organisation as Code, not as Specification

The empirical analysis conducted in sections 6.1 and 6.2 validates the central thesis that:

- **AutoGen BaseGroupChat** implements a single-group, programmatic organisation in all three dimensions: the SS is static, the FS is managed in a unified manner and the DS is implemented as a procedurally governed conversational protocol;
- **CAMEL Workforce** demonstrates strong "functional convergence" towards AOSE: it implements a more detailed and specialised SS (specialised roles, hierarchies) and an explicit FS (task decomposition as a first-class function).

However, analysis of the Deontic Dimension in both frameworks proves the paradigmatic divergence. Even in an advanced system such as CAMEL, norms (permissions and obligations) are not defined as external declarative specifications, but are implemented programmatically and merged into the procedural code.
Code analysis suggests that, in current generative frameworks, organisation is primarily programmed - defined in constructors, methods, and procedural logic - and implicit. This is a design choice that prioritises code flexibility and integration with modern programming paradigms, differing from the AOSE approach - such as Moise - that prioritises external, machine-readable declarative specifications.
The following table visually summarises the dimensional analysis.

Table 6.1: Comparative Dimensional Analysis of Organisational Implementation

Dimension	**Moise model (Ideal Specification)**	**AutoGen BaseGroupChat (Implicit Implementation)**	**CAMEL Workforce (Hybrid Implementation)**
Structural (Roles)	Declarative, specialised	Programmatic (generic list of participants)	Programmatic, specialised (Coordinator Agent, Task Planner Agent)
Structural (Hierarchy)	Declarative (Groups/Subgroups)	Non-hierarchical (single BaseGroupChat)	Programmatic (explicit hierarchy via _children, add_workforce)
Functional (Schemas)	Declarative (hierarchical breakdown)	Implicit (emerges from conversation)	Programmatic (method _decompose_task)

Continued on next page

Table 6.1 – continued from previous page

Dimension	Moise model (Ideal Specification)	AutoGen BaseGroupChat (Implicit Implementation)	CAMEL Workforce (Hybrid Implementation)
Functional (Missions)	Declarative (assigned to roles)	Implicit (single TASK: STR in run_stream)	Programmatic (Task and subtask objects)
Deontics (Norms)	Declarative (Obligations, Permissions)	Implicit (procedural governance only, e.g. termination_condition)	Programmatic (merged into the code, e.g. _create_worker _node_for_task)

6.3.1 Engineering Implications

This paradigmatic divergence, and in particular the choice of a programmatic implementation of the deontic dimension, involves different engineering trade-offs with respect to the classic objectives of robustness and governability defined by AOSE.

Origin of Verifiability

The approach to verification changes from formal verification of compliance with an external specification (typical of AOSE), to verification of procedural correctness (e.g. unit testing and integration testing of code). Since rules do not exist as artefacts separate from the execution logic, it is not possible for an external mechanism to verify whether an agent is violating an obligation or acting without permission; it is only possible to verify the correct execution of the business logic and exception handling. Therefore, in this paradigm, the separation between the specification of the rule and its execution, which is the classic AOSE approach to verifiability, is not adopted.

Approach to Organisational Reasoning

An agent is not 'organisation-aware' in the AOSE sense (it does not consult an external specification), its 'awareness' is agent-centric and delegated to the inference of the LLM model (based on prompts) and the procedural logic in which it operates: CAMEL's Coordinator Agent does not ask itself 'What are my obligations?' or 'Who has authority over me?', but deduces its behaviour through LLM inference instead of consulting a formal specification of its permissions.

Dynamic Reorganisation Model

Changing organisational rules follows a standard software development lifecycle. In a Moise-based system, changing a permission could mean updating the organisational specification at runtime. In the frameworks analysed, changing a rule (e.g. allowing a role other than Coordinator to create workers) would require a change in the source code and a redeployment. This shifts flexibility from runtime - as in AOSE systems - to design-time and deploy-time, favouring traceability of changes over immediate adaptation.

In conclusion, empirical analysis confirms that, although functional convergence on structures and functions is underway, paradigmatic divergence on deontic governance introduces different engineering trade-offs. A future synthesis, as outlined in Chapter 4.8, which could integrate the declarative approaches of AOSE with the deliberative flexibility of generative agents, represents a possible direction of research for the construction of autonomous systems that combine the deliberative intelligence of generative models with the reliability and governability of AOSE approaches.

Chapter 7

Summary, Validation and Future Prospects

This final chapter serves as a conclusion to the argument developed in this work. The aim is both to consolidate the empirical evidence gathered in Chapters 5 and 6 in order to validate the central thesis of this work, and to project the analysis into the future by proposing an engineering synthesis that addresses the trade-offs identified and outlines directions for future research.
We will proceed with a structured summary of the results, followed by a detailed response to the research questions posed in Chapter 1. Subsequently, a conceptual hybrid architecture will be proposed as a solution to the problems of paradigmatic divergence. Finally, the chapter will conclude with a discussion of the limitations of this study and the resulting prospects for future work.

7.1 Summary of Results and Thesis Validation

The research path pursued in this work began with an observation introduced in Chapter 1: the coexistence of a 'functional convergence' and a 'paradigmatic divergence' between the established principles of AOSE and emerging frameworks for generative agents. Chapters 5 and 6 provided the empirical evidence necessary to support and validate this thesis.

Functional Convergence: The 'Reinvention' of AOSE Abstractions

The comparative analysis in Chapter 5 showed that, although modern frameworks do not explicitly adopt the AOSE Agents & Artifacts (A&A) meta-model, they functionally reinvent the same abstractions, given the intrinsic nature of multi-agent problems.
Analysis of AutoGen's BaseGroupChatManager code revealed that, although implemented as an agent, its functional role is not that of a deliberative participant, but rather that of a coordination artefact. It encapsulates the state of the interaction (e.g. the list of participants, the message history), manages the lifecycle (termination and reset) and governs the interaction protocol (e.g. turn rotation). Similarly, analysis of UserProxyAgent highlighted its role as a mediation artefact, designed to abstract an external resource (the human user) and manage the architectural 'impedance mismatch' between synchronous and asynchronous I/O.
These results confirm that coordination, mediation, and shared state management issues are fundamental engineering challenges in any complex multi-agent system.

Paradigmatic Divergence: Alternative Solutions

While convergence exists at the problem level - the "what" -, divergence clearly emerges at the solution level - the "how". Empirical evidence confirms that modern frameworks adopt alternative and legitimate design choices, aligned with contemporary software development paradigms, which deviate from the canonical solutions of AOSE.
The first divergence, which emerged from Chapter 5, lies in the implementation paradigm of the environment. As summarised in Section 5.4, AOSE - in particular CArtAgO - promotes an environment-centric approach, where the artefact is a first-class, public and observable entity. Generative frameworks, on the other hand, adopt an agent-centric approach: the coordination artefact (e.g. BaseGroupChatManager) is implemented as a specialised agent. This choice introduces a

fundamental engineering trade-off: we move from an "observed state" - a public and inspectable property of the environment - to a "communicated state" - a private state of the agent-manager, which other participants can only infer from the messages they receive. This directly impacts the observability, verifiability, and debugging of the system.
The second and most profound divergence, analysed empirically in Chapter 6, concerns organisational governance: dimensional analysis with respect to the Moise model revealed that, although systems such as CAMEL show structural convergence - defining explicit roles and hierarchies -, implementation remains programmatic and implicit.
The critical conclusion of Chapter 6 is that, in these frameworks, organisation is treated as "Code, not Specification". The divergence is greatest in Deontic Dimension where norms (permissions, obligations) are not declarative constructs, external and consultable at runtime - as in Moise - but are instead fused into procedural logic. As highlighted in the CAMEL analysis, the "permission" of a Coordinator agent to create a new worker is not a consultable rule, but is implicitly defined by the very existence of the create_worker_node_for_task method.

Thesis Validation

Based on established empirical evidence, the central thesis of this work, formulated in Section 1.3, has been validated. Generative agent frameworks are indeed addressing the same fundamental problems of coordination, mediation, and governance for which AOSE has been developing formal solutions for decades - the 'functional convergence'. However, they do so through alternative and legitimate design choices - an agent-centric, programmatic, and implicit approach - that represent a clear 'paradigmatic divergence'. This divergence favours prototypical flexibility and alignment with modern software development cycles, while introducing a new set of engineering trade-offs related to observability, verifiability, and separation of responsibilities.

7.2 Answering Research Questions

This section provides an explicit and consolidated answer to each of the research questions posed in Section 1.4, summarising the findings of Chapters 2, 3, 5 and 6.

Q1) *To what extent do modern frameworks for generative agents implicitly converge, diverge, or reinvent the founding principles of AOSE, and what are the engineering implications of this relationship?*

Answer: Modern frameworks show strong functional convergence and clear paradigmatic divergence: they reinvent the need for environmental abstractions for coordination - as demonstrated in Chapter 5 - and governance structures - as highlighted in Chapter 6 -, confirming the universality of AOSE principles. However, they diverge in implementation, favouring an agent-centric and programmatic approach ("Organisation as Code") over the environment-centric and declarative approach of AOSE (CArtAgO, Moise). The engineering implications, discussed in sections 5.4 and 6.3.1, lie in a conscious trade-off: modern frameworks trade formal observability, deontic verifiability and strict separation of responsibilities - typical of AOSE - for faster prototyping and easier integration with current software development paradigms - as mentioned in section 1.3.

Q2) *To what extent do modern prompting mechanisms for emergent reasoning, such as Chain-of-Thought and ReAct, functionally emulate the practical reasoning cycle of the Belief-Desire-Intention (BDI) model? What structural guarantees do the two approaches offer?*

Answer: Mechanisms such as ReAct functionally emulate the BDI cycle through an iterative loop of Thought-Action-Observation. The structural guarantees offered are, however, diametrically opposed. The BDI model offers an explicit structural guarantee: its deliberative cycle is a formal, inspectable and verifiable interpreter, although its knowledge is limited and pre-coded ('rich in structure, poor in knowledge'). The LLM-based generative model offers no structural guarantees: the 'Thinking' step is an opaque call to a stochastic component, but it draws on vast implicit knowledge ('knowledge-rich, implicit structure'). The reliability of the system shifts from the formal correctness of the interpreter (BDI) to the robustness of the prompt and error handling (ReAct).

Q3) *What are the engineering implications of transforming the 'plan' from an explicit and verifiable data structure, as defined in AOSE, to an emergent 'narrative artefact', generated linguistically by LLM agents?*

Answer: The main engineering implication is a shift from a priori verifiability to a posteriori validation. A plan in a classic BDI system is a data structure that can be analysed and validated statically before execution. A 'narrative plan' generated by an LLM is inherently flexible and capable of adapting to previously unseen contexts, but it is also stochastic and non-deterministic. It cannot be verified, it can only be executed and its result observed: this shifts the burden of engineering from designing robust plans to managing failure and correcting generated plans in real time.

Q4) *How does the paradigm of the environment as a passive 'tool library', typical of generative frameworks, compare with the environment as a first-class abstraction, mediator and governor of the A&A paradigm and platforms such as CArtAgO, and what are the implications of these two choices on the modularity, observability and robustness of interaction protocols?*

Answer: The environment as a "tool library" is a passive paradigm that leads to the implementation of coordination and mediation logic within agents or, as demonstrated in Chapter 5, within specialised manager agents. The A&A/CArtAgO abstraction, on the other hand, is an active abstraction that decouples this logic and places it in the environment itself. The implications, summarised in Section 5.4, are straightforward: the CArtAgO approach offers greater advantages in terms of modularity and reusability (the artefact is independent of agents) and formal observability (the "observed state"). The 'tool library' approach offers faster prototyping but tightly couples the coordination logic to the agent implementation and makes the interaction state private and only 'communicated', impacting robustness and external inspection capabilities.

Q5) *How does the implicit and programmatic definition of organisation in generative agent systems introduce an alternative governance model (e.g. implicit or programmatic), which differs significantly from AOSE approaches based on formal specifications such as the Moise model?*

Answer: The empirical analysis in Chapter 6 confirmed that the programmatic approach is an alternative and legitimate model of governance, but that it differs from Moise in one fundamental aspect: the absence of a declarative deontic dimension. In Moise, rules (obligations, permissions) are first-class specifications that can be consulted and interpreted at runtime. In the frameworks analysed, rules are encoded as procedural logic (e.g. an if-then in the code) or as programmatic capabilities (e.g. the existence of a method). The engineering implication is the absence of organisational reasoning capability at runtime - an agent cannot "ask" what its obligations are - and formal verifiability of social norms, in exchange for greater flexibility of development and integration with native programming language constructs.

7.3 Towards an Engineering Synthesis

After critically analysing emerging architectures and validating the thesis of paradigmatic divergence, this section takes on a proactive tone. The analyses in Chapters 5 and 6 revealed that modern frameworks, while effective, introduce significant trade-offs in terms of observability and formal governance. On the other hand, classic AOSE frameworks, while robust, are rigid and place a high cognitive load on modelling.
The future, as suggested in Chapter 1, does not lie in choosing one paradigm over another, but in pursuing a rigorous engineering synthesis. Based on the integration patterns hypothesised in section 4.8, a conceptual hybrid architecture - 'JaCaMo-Gen' - is proposed, which integrates the deliberative flexibility of generative models (Gen) with the structural, environmental and organisational robustness of the JaCaMo (Jason + CArtAgO + Moise) AOSE platform.
The idea behind this hybrid architecture was developed specifically to try to resolve the trade-offs identified:

The BDI Agent (Jason) as Structured Scaffolding

The JaCaMo-Gen architecture uses the BDI agent (Jason) as structured scaffolding to directly address the challenge posed by the opaque, stochastic, and structurally unguaranteed nature typical

of deliberation based solely on LLM. In the hybrid approach, in fact, the primary computational entity is not the LLM, but rather a Jason-based BDI agent. This results in a robust architectural framework equipped with an explicit, inspectable, and verifiable practical reasoning cycle that governs the entire decision-making process.

The LLM (Gen) as a Deliberative Component 'Call-out'

Hybrid architecture addresses the challenge of rigidity and limited knowledge, typical of classic BDI systems based on libraries of static, pre-programmed plans. To overcome this limitation, the JaCaMo-Gen solution implements the 'LLM as a Deliberative Component' pattern. In this model, the LLM does not act as the primary agent, but operates as a specialised component that the BDI agent can invoke within its reasoning cycle. This allows, for example, a BDI plan to query the LLM to dynamically generate a sequence of actions (a 'narrative plan') when faced with a goal for which it has no predefined plan, or to receive assistance in evaluating complex options (such as 'which desire should I pursue in this contextual situation?').

The Explicit Environment (CArtAgO) for Robust Interaction

Engineering synthesis focuses on robust interaction, addressing the challenges introduced by the "tool library" approach and the resulting trade-off between "reported state" and "observed state," identified in Chapter 5. To solve this problem, JaCaMo-Gen implements the AOSE pattern artifacts as 'Smart Tools', which means that hybrid agents (BDI+LLM) do not interact with passive APIs, but operate directly with active CArtAgO artifacts. For example, instead of relying on a specialised agent such as BaseGroupChatManager, the hybrid architecture would use a CoordinationArtifact CArtAgO. Consequently, the state of interaction - such as 'whose turn it is' - is no longer a private communicated state, but becomes a public observable property of the environment, thus reintroducing formal observability and environmental modularity that are not present in the purely agent-centric approach.

Declarative Governance (Moise) for Social Verifiability

Finally, hybrid architecture addresses governance by using Moise for social verifiability. This solution responds directly to the 'Organisation as Code' approach and the absence of an explicit deontic dimension that were identified in Chapter 6. To achieve this goal, the 'Moise for the Governance of Generative Agent Societies' pattern is implemented: in this way, the entire society of hybrid agents operates within a specific Moise organisation, which is both explicit and declarative. Unlike a "permission" simply encoded in a method, the Moise specification formally defines roles, missions, and norms (such as permissions and obligations). This approach restores the fundamental organisational reasoning ability to the BDI agent (for example, allowing it to ask itself "As a 'Tester', am I allowed to perform the 'deploy' action in this schema?"), effectively reintroducing deontic verifiability into the system.

In summary, the JaCaMo-Gen architecture does not discard generative frameworks, but encapsulates them within a robust engineering framework, using LLM for what it does best - flexible knowledge generation - and AOSE for what it does best - providing structure, observability, and verifiable governance.

7.4 Limitations of the Study and Future Work

This work concludes with a critical reflection on the limitations of the present analysis and on the lines of research that arise from it.
Despite attempts to conduct a rigorous analysis, this study has two inherent limitations:

- **Scope of the Analysis**
  The empirical analysis was conducted on a representative but limited subset of frameworks (primarily AutoGen, Eclipse LMOS Arc and CAMEL, as outlined in Section 4.1). Although the results are indicative of broader paradigmatic trends, they cannot be generalised to the entire, heterogeneous ecosystem of generative agents without further comparative studies;

- **Temporal Context**
  The field of generative agents is characterised by extremely rapid technological evolution, described in Section 4.7.3 as a 'continuous integration race': the frameworks analysed are a

"moving target". This thesis provides a critical analysis and a snapshot that is valid at the present time, but its specific conclusions on individual frameworks may need to be updated as they evolve, potentially integrating AOSE principles more explicitly in the future.

The conclusions and proposed summary of this work pave the way for two main and complementary areas of future research.

Implementation and Evaluation of Hybrid Architecture

The most immediate logical consequence of the entire work is the implementation and empirical evaluation of the JaCaMo-Gen hybrid architecture proposed in Section 7.3. This future work would be essential to experimentally verify the hypotheses formulated. This evaluation should measure, through quantitative (e.g. fault tolerance, recovery times) and qualitative (e.g. observability, ease of debugging) metrics, whether the proposed synthesis effectively resolves the identified trade-offs, offering a measurable improvement over the use of pure JaCaMo - in terms of flexibility - or pure generative frameworks - in terms of robustness and governability.

Automation of AOSE Modelling through LLM

This future work closes the circle, using the strengths of the generative paradigm to mitigate the historical limitations of the AOSE paradigm. As discussed in Section 1.3, a significant obstacle to the adoption of AOSE is the high 'cognitive load' required for formal specification. As hypothesised in Section 4.8 1, a research direction is the development of LLM-based tools for so-called 'AOSE Modelling Automation'. These tools would allow a developer to describe organisational requirements (e.g. 'I need a code review team with one senior reviewer and two junior reviewers; the senior reviewer has final authority') or environmental requirements (e.g. 'I want an artefact for a Dutch auction') in natural language. The LLM would act as an 'assistant AOSE engineer', generating drafts of formal Moise specifications or CArtAgO artefact templates. This approach would help lower the barrier to entry for robust engineering, facilitating the large-scale adoption of robust hybrid architectures such as JaCaMo-Gen.

Bibliography

[Bar24] Saikat Barua. Exploring autonomous agents through the lens of large language models: A review. *arXiv preprint arXiv:2404.04442*, 2024.

[Fou] Eclipse Foundation. Lmos arc. `https://eclipse.dev/lmos/arc2/index.html`.

[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. *arXiv preprint arXiv:2402.01680*, 2024.

[HSB02] Jomi Fred Hübner, Jaime Simao Sichman, and Olivier Boissier. Moise+ towards a structural, functional, and deontic model for mas organization. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 501–502, 2002.

[HSB07] Jomi F Hubner, Jaime S Sichman, and Olivier Boissier. Developing organised multi-agent systems using the moise+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering*, 1(3-4):370–395, 2007.

[HTL25] Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

[LHI+23] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

[MBSC24] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint arXiv:2404.11584*, 2024.

[MDL+23] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

[ORV+04] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 286–293, 2004.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous agents and multi-agent systems*, 17(3):432–456, 2008.

[POC+23] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[RNI95] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25(27):79–80, 1995.

[RPVO09] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. Environment programming in cartago. In *Multi-agent programming: Languages, tools and applications*, pages 259–288. Springer, 2009.

[SYNG23] Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2023.

[WBZ+24] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

[Wen23] Lilian Weng. Llm-powered autonomous agents. *lilianweng.github.io*, Jun 2023.

[WG11] Danny Weyns and Marie-Pierre Gleizes. *Agent-Oriented Software Engineering XI: 11th International Workshop, AOSE XI, Toronto, Canada, May 10-11, 2010, Revised Selected Papers*, volume 6788. Springer Science & Business Media, 2011.

[WMF+24] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[WOO07] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous agents and multi-agent systems*, 14(1):5–30, 2007.

[Woo09] Michael Wooldridge. *An introduction to multiagent systems*. John wiley & sons, 2009.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[XCG+25] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[YZY+22] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.