University of Zagreb

Faculty of Organization and Informatics

Tomislav Peharda

PROGRAMMING LANGUAGE FOR COMMUNICATION FLOWS SPECIFICATION IN MULTI-AGENT SYSTEMS

DOCTORAL THESIS

Varaždin, 2025

Sveučilište u Zagrebu

Fakultet organizacije i informatike

Tomislav Peharda

PROGRAMSKI JEZIK ZA SPECIFIKACIJU KOMUNIKACIJSKIH TOKOVA U VIŠEAGENTNIM SUSTAVIMA

DOKTORSKI RAD

Varaždin, 2025.

University of Zagreb

Faculty of Organization and Informatics

Tomislav Peharda

PROGRAMMING LANGUAGE FOR COMMUNICATION FLOWS SPECIFICATION IN MULTI-AGENT SYSTEMS

DOCTORAL THESIS

Supervisor:
Full Prof. Markus Schatten

Varaždin, 2025

Sveučilište u Zagrebu

Fakultet organizacije i informatike

Tomislav Peharda

PROGRAMSKI JEZIK ZA SPECIFIKACIJU KOMUNIKACIJSKIH TOKOVA U VIŠEAGENTNIM SUSTAVIMA

DOKTORSKI RAD

Mentor:
Prof. dr. sc. Markus Schatten

Varaždin, 2025.

Markus Schatten is a full professor and the head of the Artificial Intelligence Laboratory as well as the head of the Doctoral study programme Information sciences at the Faculty of Organization and Informatics, University of Zagreb. He has defended his PhD thesis in the field of information and communication sciences, has a master thesis in information sciences and a diploma in information systems. He authored over 110 scientific and professional papers and two books. He is holding lectures on all university degrees (bachelor, master, specialist, doctoral) in the fields of artificial intelligence, computer games and related fields. His field of research includes artificial intelligence and its application to various domains including but not limited to computer games, education, multiagent systems, virtual assistants, the Internet of Everything and others.

He has participated in about 15 scientific and professional projects including a few international ones (FP7, COST, Erasmus+, EEAGrants). He has lead a scientific installation project (ModelMMORPG Large-scale Multi-agent Modeling of Massively Multiplayer on-line Role Playing Games), as well as a research project (O_HAIGames - Orchestration of Hybrid Artificial Intelligence Methods for Computer Games) both sponsored by the Croatian Science Foundation.

Abstract

Abstract in English

Multi-agent systems (MASs) are valuable in performing complex tasks that require autonomy. As the number of agents in such a system increases, more levels of complexity must be addressed to enable coordinated functioning. One of the challenges is the specification of communication flows. Agents exchange messages via communication channels to inform each other about their requests (for data, task execution, or similar) or their current status (whether they are processing or ready to take on a task). Therefore, specifying communication flows is crucial for achieving cohesion between agents. Since agents are autonomous and independent units, traditional design approaches involve implementing communication flows and business logic individually for each agent. This leads to redundant, unstable, less readable, and less extensible code, particularly when agents within a MAS are considered as a whole. A proposed solution to this problem is the construction of a programming language for specifying communication flows, based on process calculus, coupled with the development of a declarative engine. This engine is capable of processing these specifications and coordinating their execution. This programming language allows engineers to define communication flows between agents, ensuring consistency in communication and enabling orchestration within the MAS
Keywords. multi-agent systems, orchestration, intelligent agents

Abstract in Croatian

Višeagentni sustavi (VAS) korisni su u izvođenju kompleksnih zadataka koji iziskuju autonomnost. Što je broj agenata u nekom sustavu veći, to je veći broj razina kompleksnosti koje je potrebno uzeti u obzir kako bi se omogućilo koordinirano funkcioniranje sustava. Jedan od izazova je i specifikacija komunikacijskih tokova. Agenti razmjenjuju poruke putem komunikacijskih kanala kako bi se međusobno obavijestili o zahtjevima (o podacima, rješavanju zadataka, ili slično) ili pak njihovom statusu (izvršavaju li trenutno neki zadatak, ili su slobodni za novi zadatak). Stoga je specifikacija komunikacijskih tokova važna za ostvarenje kohezije u radu agenata. S obzirom da su agenti autonomne i nezavisne jedinke, tradicionalni pristupi implementaciji komunikacijskih tokova i poslovne logike implementiraju iste za svakog agenta zasebno. To dovodi do redundantnosti kôda, nestabilnosti, smanjene čitljivosti te slabe proširivosti, posebice kada se agenti u višeagentnom sustavu promatraju kao cjelina. Predloženo rješenje za ovaj problem jest u razvoju programskog jezika za specifikaciju komunikacijskih tokova baziranom na procesnom računu te razvoju deklarativnog stroja. Deklarativni stroj zaslužan je za procesuiranje specifikacija te koordinaciju njihovog izvršavanja. Programskih jezik omogućuje da inženjer specificira komunikacije tokove među agentima što rezultira konzistentnost u komunikaciji

te mogućnost orkestracije agenata u VAS-u.

Ključne riječi. višeagentni sustavi, orkestracija, inteligentni agenti

Contents

List of Used Acronyms

MAS	Multi-agent system
SPADE	Smart Python Agent Development Environment
AI	Artificial Intelligence
ML	Machine Learning
Gen AI	Generative Artificial Intelligence
XMPP	Extensible Messaging and Presence Protocol
FSM	Finite State Machine
ANTLR	ANother Tool for Language Recognition
O HAI 4 Games	Orchestration of Hybrid Artificial Intelligence Methods for Computer Games
MMORPG	A Massively Multiplayer Online Role-playing Game
CV	Computer Vision
XMPP	Extensible Messaging and Presence Protocol
FSM	Finite State Machine
DSR	Design Science Research
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
CS	Computer Science
LLM	Large Language Model
GAN	Generative Adversarial Network
AWS	Amazon Web Services
GCP	Google Cloud Platform
IoT	Internet of Things
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
CFG	Context-free grammar
JADE	Java Agent DEvelopment Framework
FIPA	Foundation for Intelligent Physical Agents
ACL	Agent Communication Language
IF	Interactive Fiction

MMO	Massively Multiplayer Online
NPC	Non-player character
B.A.R.I.C.A	Beautiful Artificial Intelligence Cognitive Agent
FEDS	Framework for Evaluation in Design Science research
BDI	Belief-Desire-Intention
TTS	Text-to-Speech
STT	Speech-to-Text
NLP	Natural Language Processing
HMAS	Holonic Multiagent System
DT	Digital Twin
REST	Representational State Transfer
WS	WebSocket
laGGer	Lag but Good Game
DL	Deep Learning
YAML	Yet Another Markup Language
AV	Autonomous Vehicle
JSON	JavaScript Object Notation
XML	Extensible Markup Language
HTTP	Hypertext Transfer Protocol
REGEX	Regular Expression
GPS	Global Positioning System
SSE	Server-Sent Events

List of Figures

Listings

Prošireni sažetak na hrvatskom jeziku

Uvod i pregled dosadašnjih istraživanja

Višeagentni sustav (VAS) može se opisati kao klaster većeg broja međusobno povezanih agenata koji zajedničkim djelovanjem nastoje doći do rješenja zadanog problema, kojeg pojedinačno nebi mogli ostvariti. S obzirom na karakteristike autonomnosti te interakcije s drugim jedinkama, agent se također može promatrati kao heterogeni mikroservis [122, 58]. Agenti u VAS-u komuniciraju kako bi razmijenili informacije koje bi drugoj strani mogle biti korisne u obavljaju svojih zadataka. Iz tog razloga, orkestracija agenata postaje uvelike važna za postizanje očekivanih ishoda [54]. U slučaju gdje veći broj VAS-a sudjeluje u komunikaciji, često se koristi koncept holonskih sustava. Spomenuti koncept predlaže da postoji reprezentativni agent VAS-a (holon) koji je odgovoran za komunikaciju s vanjskim holonima te isto tako prosljeđuje poruke odgovarajućim agentima unutar svog holona [25]. Budući da je agent samostalna i neovisna jedinka, za svaku jedinku se komunikacijski mehanizmi implementiraju pojedinačno što uzrokuje redundanciju programskog kôda, manju stabilnost, manju čitljivost te proširivost pri opservaciji cjelokupnog VAS-a što dovodi do otežane orkestracije agenata [54]. U kontekstu ovog istraživanja, orkestracija agenata odnosi se na konzistentnu specifikaciju komunikacijskih tokova između parova agenata, čime se implicira da je svaki agent dizajniran za međusobnu komunikaciju.

Prethodno opisani izazovi u dizajnu komunikacijskih tokova u VAS-u primarni su motiv za provođenje ovog istraživanja. Kako bi se poduprlo istraživanje, analizirane su dvije domene literature: znanstveni radovi te prakse iz industrije. Ove domene prepoznate su kao relevantne jer uključuju pristupe koji se razmatraju sa znanstvenog stajališta, ali i pristupe slučajeva korištenja iz realne domene.

Scopus [27], WoS [21] i IEEE Xplore [43] su baze podataka koje su pretražene s ciljem pronalaska znanstvenih radova. Analizirani su radovi koji imaju najveći broj citata u domeni komunikacijskih tokova VAS-a. Primjerice, Corkill [24] predlaže koncept emitiranja poruka, gdje svaki agent šalje poruke svim ostalim agentima u VAS-u, pri čemu svaki agent samostalno implementira mehanizme filtriranja poruka koje za njega nisu relevantne. Ovaj pristup donosi fleksibilnost u dizajnu i implementaciji agenata, ali ne uzima

u obzir smanjene performanse zbog povećanog prometa poruka. Goldman i Zilberstein [33] nadograđuju ideju na konceptu emitiranja poruka, uzimajući u obzir činjenicu da eksterni agenti (izvan promatranog VAS-a) vjerojatno nisu prilagođeni za filtriranje poruka koje im nisu relevantne. Stoga predlažu ograničavanje broja agenata unutar VAS-a koji komuniciraju s vanjskim agentima. Nadalje, svi agenti unutar VAS-a komunicirali bi isključivo s reprezentativnim agentima VAS-a, koji bi potom prosljeđivali poruke odgovarajućim vanjskim agentima [33]. Ova ideja slična je konceptu holonskog sustava [25]. Rezultat ovog pristupa je ograničen broj agenata unutar VAS-a koji implementiraju pouzdanu logiku za komunikaciju u kontekstu prosljeđivanja poruka. Ostatak analiziranih radova bave se utjecajem različitih algoritama na pronalazak optimalne rute komunikacije između izvornog i odredišnog agenta, u slučajevima kada izravna komunikacija nije moguća te su potrebni posrednički agenti [24, 33, 3]. Iako se analizirani radovi bave problematikom razmjene poruka među agentima, njihova su rješenja ograničena na tradicionalni dizajn i implementaciju za konkretne probleme komunikacije, bez nastojanja da se ponudi univerzalno rješenje za specifikaciju komunikacijskih tokova.

Literatura iz domene praksi iz industrije fokusirana je na platforme orkestracije sustava. Agent se također može promatrati kao takav sustav budući da obavlja unaprijed definirane zadatke kako bi proizveo rješenje. U načelu, svaki agent pruža barem jedan servis, no svaki zaseban servis ne može se nužno smatrati agentom ako nije dovoljno autonoman u obavljanju zadatka [26]. Problem komunikacijskih tokova unutar sustavno orijentiranih arhitektura (posebice u kontekstu mikroservisa u suvremenim okuženjima) naziva se orkestracija [127]. Na tržištu postoji nekolicina platformi za orkestraciju sustava, no one obično omogućuju orkestraciju na visokoj razini apstrakcije, pri čemu ne uzimaju u obzir specifičnu prirodu samog sustava. Primarni fokus spomenutih platformi odnosi se na pokretanje servisa u pravilnom redoslijedu unutar istog klastera sustava, čime se omogućuje lakša suradnja među servisima te ravnomjernija raspodjela zahtjeva po pojedinom sustavu [90, 91]. Kubernetes [18] je platforma otvorena kôda koja omogućuje orkestraciju sustava pomoću deklarativne konfiguracije. Jedna od glavnih značajki platforme jest mogućnost skaliranja klastera sustava te automatizacije procesa uključenja u rad. Kubernetes ima vlastiti ekosustav za potporu zajedničkog rada sustava unutar klastera u kojem su stacionirani. No, Kubernetes ne nudi rješenje za komunikaciju među sustavima. U Kubernetesovom ekosustavu, sustav je izoliran u zasebno okruženje s pristupom resursima klastera u kojem se nalazi. Međutim, karakteristike specifične za pojedinu vrstu sustava moraju biti zasebno projektirane i implementirane. To ukazuje na to da ova platforma ne nudi rješenja prilagođena svakom pojedinom sustavu, već djeluje isključivo na visokoj razini apstrakcije [90]. Docker Swarm [106] je način rada platforme Docker koja pruža alate za orkestraciju sustava slične Kubernetesu. Razlike između ove dvije platforme proizlaze iz njihove definicije ekosustava, pri čemu Docker Swarm nudi ograničeniji skup funkcionalnosti u pogledu konfiguracije infrastrukture i pristupa povezi-

vanju većeg broja klastera. Docker Swarm također ne nudi rješenje za pojednostavljenje komunikacije među sustavima [106, 85]. Postoji nekolicina alternativnih platformi orkestracije sustava kao što su Azure Container Instance [60], Google Cloud Run [36], itd.. Unatoč tome, nijedna od istraženih platformi ne bavi se specifikacijom komunikacijskih tokova. Njihov je glavni fokus pružanje infrastrukture u kojoj su sustavi smješteni [85]. Prethodno opisana razina apstrakcije koju ove platforme nude opravdana je s obzirom na to da su mehanizmi komunikacije specifični za svaki pojedini sustav. Stoga je iznimno složeno razviti univerzalno rješenje za sve vrste komunikacijskih tokova, bez obzira na vrstu sustava. Većina analiziranih platformi orkestraciju temelji na konfiguracijskim datotekama, pri čemu konfiguracija omogućuje alokaciju resursa, definiranje međuzavisnosti sustava i slične postavke. Iz tog razloga, proširenje konfiguracijskih funkcionalnosti radi specifikacije komunikacijskih tokova nije izvedivo.

Analizom literature utvrđeno je da postoji značajan prostor za poboljšanje u pogledu specifikacije komunikacijskih tokova unutar VAS-a. Istraženi znanstveni radovi uglavnom se bave modeliranjem agenata u okviru ograničenja samog VAS-a. S druge strane, postojeće platforme za orkestraciju sustava primarno su usmjerene na apstrahiranje zajedničkih karakteristika sustava poput resursa, infrastrukture i sličnih elemenata, neovisno o njihovoj vrsti. Budući da se ovim istraživanjem predlaže razvoj novog artefakta temeljenog na procesnom računu, valja istaknuti da, osim programskog jezika Pict, nisu pronađeni drugi široko poznati programski jezici koji se temelje na toj paradigmi [80].

Ciljevi istraživanja

Primarna istraživačka pitanja na koja je usredotočen ovaj rad usmjerena su na konstrukciju programskog jezika za orkestraciju agenata u višeagentnoj arhitekturi koji zadovoljava potrebe modernih domena te omogućuje inženjeru jednostavnu specifikaciju komunikacijskih tokova. Shodno tome, istraživačka pitanja su sljedeća:

IP1 Koje vrste komunikacijskih tokova treba podržati programski jezik s obzirom na potrebe modernih domena vezanih uz mikroservisnu arhitekturu, umjetnu inteligenciju i računarstvo u oblaku?

IP2 Kako oblikovati sintaksu, semantiku i pragmatiku programskog jezika za orkestraciju heterogenih mikroservisa u višeagentnoj arhitekturi koristeći procesni račun?

IP3 Na koji način podržati oblikovanje ansambla kompleksnih metoda koristeći holonske sustave?

Istraživački ciljevi fokusirani su na razvoj artefakata koji će omogućiti pojednostavljanjenje orkestracije agenata u VAS-u kroz programski jezik te deklarativni stroj. Iz toga proizlaze slijedeći ciljevi:

C1 Razviti programski jezik koji će omogućiti orkestraciju heterogenih mikroservisa u višeagentnoj arhitekturi

C2 Razviti deklarativni stroj temeljen na procesnom računu koji će biti u stanju upravljati komunikacijskim tokovima među inteligentnim agentima

Nadalje, hipoteza koja se ispituje ovim radom jest:

H1 Programski jezik za specifikaciju komunikacijskih tokova temeljen na procesnom računu unaprijedit će orkestraciju heterogenih mikroservisa koristeći višeagentnu arhitekturu

Metodologija

Ovim istraživanjem razvijeni su artefakti koji nude rješenje za prepoznate nedostatke povezane sa specifikacijom komunikacijskih tokova u VAS-u. Stoga je odgovarajuća istraživačka paradigma za ovo istraživanje pragmatizam. Budući da je istraživanjem konstruirano novo rješenje koje utječe na način na koji softverski inženjeri definiraju komunikacijske tokove te na njihovo iskustvo implementacije u usporedbi s tradicionalnim pristupima dizajnu takvih sustava, istraživanje slijedi kvalitativni pristup. Metodologija koja najbolje odgovara postavljenim ciljevima i očekivanim rezultatima jest znanost o dizajnu [119, 77]. U okviru ovog istraživanja definiran je skup zahtjeva koje je artefakt morao zadovoljiti tijekom implementacije. Artefakt je isključivo usmjeren na poboljšanje komunikacijskih tokova među agentima, dok ostale funkcionalnosti agenata nisu bile predmet ovog istraživanja. Tijekom evaluacijske faze analizirana je ispunjenost postavljenih zahtjeva te su identificirane prednosti i nedostaci artefakta na temelju povratnih informacija evaluatora koji su sudjelovali u njegovom testiranju.

Primarne metode korištene u ovom istraživanjau uključuju formalizaciju komunikacije agenata unutar VAS-a korištenjem procesnog računa [105, 7], modeliranje programskog jezika na temelju unaprijed definiranih vrsta komunikacijskih tokova [14] te razvoj programskog jezika i deklarativnog stroja primjenom metoda programskog inženjerstva [26]. Sintaksa jezika formalizirana je korištenjem kontekstrualno slobodnih gramatika i apstraktnih stabala sintakse. Semantika jezika slijedi operativnu semantiku π-računa [76] koji je instanca procesnog računa. U svrhu formalizacije komunikacije agenata, prijedlog je primijeniti procesni račun. Procesni račun predstavlja obitelj pristupa za modeliranje konkurentnih sustava s fokusom na opisivanje interakcije, komunikacije i sinkronizacije agenata. Glavne značajke svih pristupa procesnom računu uključuju: prijenos poruka između neovisnih procesa koji predstavljaju interakcije, ograničen broj elemenata i operatora za opis procesa te specifična algebarska pravila za njihove operatore. Pravilo redukcije predstavlja ključni aspekt procesnog računa jer omogućuje jednoznačan opis komunikacije, uzimajući u obzir paralelnu kompoziciju, sljednost i transformaciju ulaza u izlaz [105,

7]. Procesni račun daje formalnu pozadinu za modeliranje složenih distribuiranih sustava i koristi se kao teoretska osnova za istraživanje.

Cilj metode modeliranja jest oblikovati karakteristike i elemente novog programskog jezika na temelju koncepata razvoja komunikacijskih tokova u VAS-u [14] te koncepata holonskih sustava u prethodno definiranim granicama [105, 7]. Ključni koncepti korišteni tijekom implementacije usmjereni su na specifikaciju komunikacijskih tokova u skladu s unaprijed definiranim zahtjevima. Holoni predstavljaju prikladan koncept za modeliranje kompleksnih distribuiranih sustava jer omogućuju organizaciju sustava u manje jedinice (holone) kroz hijerarhijsku strukturu [25]. U skladu s prethodno konstruiranim modelom jezika, programski jezik i deklarativni stroj razvijeni su primjenom metoda programskog inženjerstva [26]. Programski jezik koristi se za konstrukciju specifikacije komunikacijskih tokova, odnosno orkestracije agenata, dok je uloga deklarativnog stroja procesuiranje te specifikacije.

U prvoj fazi odabrane metodologije provedena je analiza literature o temama vezanim uz VAS, mikroservisnu arhitekturu, umjetnu inteligenciju, orkestraciju i računarstvo u oblaku, s posebnim fokusom na aspekt specifikacije komunikacijskih tokova. Kvalitativnom analizom karakteristika postojećih pristupa orkestraciji identificiran je prostor za unaprjeđenje. Osim znanstvene literature, detaljno su istražene i postojeće platforme za orkestraciju sustava, s ciljem utvrđivanja njihovih primarnih funkcionalnosti i industrijskih praksi. Nakon dovršetka analize i identifikacije problema, pristupilo se sljedećoj fazi istraživanja koja se bavi prikupljanjem zahtjeva za dizajn artefakta i izradom plana rada s jasno definiranim ciljevima. U skladu s time, zahtjevi za dizajn artefakta definirani su na temelju analizirane znanstvene literature, industrijskih standarda te ciljeva projekta O HAI 4 Games (Orkestracija hibridnih metoda umjetne inteligencije s primjenom na računalne igre) [100]. U ovoj fazi jasno su definirane vrste komunikacijskih tokova koje programski jezik podržava, uzimajući u obzir ograničenja razvoja agenata u VAS-u te potrebe holonskih sustava. Opis pristupa primjene procesnog računa u specifikaciji komunikacijskih tokova dan je po završetku ove faze te je korišten u sljedećoj fazi, pri dizajnu i razvoju artefakta, odnosno implementaciji programskog jezika. U fazi dizajna i razvoja artefakta, cilj je razviti programski jezik u skladu s unaprijed definiranim zahtjevima, temeljem prethodno izrađenog plana rada. Proces je podržan ranije opisanim metodama formalizacije, modeliranja i programskog inženjerstva. Ova faza obuhvaća dizajn sintakse, semantike i pragmatike jezika, kao i implementaciju deklarativnog stroja koji je sposoban procesuirati specifikaciju komunikacijskih tokova koje pruža korisnik (programer). U četvrtoj fazi metodologije, novorazvijeni artefakt testiran je i evaluiran na slučajevima korištenja u stvarnim domenama. Slučajevi korištenja implementirani su kroz projekt O HAI 4 Games, u kontekstu masivnih online igara (MMORPG), kognitivnih agenata u gamifikaciji, ozbiljnih igara te holografskih igara [100].

Po prijelazu u fazu evaluacije rješenja, artefakt je evaluiran kroz provjeru može li

programski jezik adekvatno podržati unaprijed definirane vrste komunikacijskih tokova, potvrdu ispravnosti semantičke analize te detaljnu procjenu od strane projektnog tima O HAI 4 Games i nezavisnih stručnjaka iz područja računarstva u oblaku, umjetne inteligencije i mikroservisne arhitekture. Proces evaluacije proveden je u skladu s FEDS okvirom, koji se sastoji od četiri koraka: (1) definiranje ciljeva evaluacije, (2) odabir strategije evaluacije, (3) određivanje evaluacijskih karakteristika, i (4) oblikovanje pojedinačne iteracije evaluacije [116].

Evaluacija ima za cilj procijeniti sposobnost softverskog artefakta za učinkovito upravljanje komunikacijskim tokovima u informacijskim sustavima, s fokusom na kompatibilnost, modularnost, iskoristivost, performanse i pridržavanje kriterija programskih jezika poput čitljivosti, jednostavnosti pisanja i pouzdanosti. Strategija evaluacije uključuje formativne i sumativne metode. Formativna evaluacija provodit će se u kontroliranim uvjetima, koristeći simulacije i strukturirane povratne informacije s ciljem postupnog unaprjeđenja artefakta. S druge strane, sumativna evaluacija procjenjivat će integraciju artefakta u stvarnim uvjetima, kroz praktične slučajeve i prikupljanje povratnih informacija od članova tima i stručnjaka putem promatranja, intervjua i anketa [116].

Karakteristike evaluacije definirane su na temelju standarda ISO/IEC 25002:2024 te kriterija za procjenu programskih jezika. Evaluacijske epizode obuhvaćaju implementaciju artefakta u stvarnim slučajevima, prikupljanje povratnih informacija, kvalitativnu analizu prikupljenih podataka i izradu sveobuhvatnog izvješća s preporukama za daljnje poboljšanje [114, 111, 84].

Struktura disertacije podijeljena je u devet poglavlja, povezanih s fazama metodologije znanosti o dizajnu [41]. U 1. poglavlju predstavljeni su motivacija za provođenje istraživanja, opis relevantnih domena i područja, te pregled srodne znanstvene i stručne literature. Slijedi 2. poglavlje koje opisuje ključne koncepte, definicije, arhitekture i pripadajuće alate, s ciljem boljeg razumijevanja temelja istraživanja. U 3. poglavlju provedena je analiza postojećih razvojnih okvira i knjižnica za razvoj VAS-a. Poglavlje 4 donosi niz zahtjeva za razvoj artefakta, čija je implementacija razrađena u 5. poglavlju. Poglavlje 6 prikazuje implementaciju artefakta na slučajevima korištenja u sklopu projekta O HAI 4 Games. Nadalje, poglavlje 7 bavi se evaluacijom provedenom nad artefaktom. Na kraju, poglavlja 8 i 9 daju uvid u moguća poboljšanja te završne zaključke.

Chapter 1

Introductory notes

1.1 Motivation

With the recent innovations in the field of Artificial Intelligence (AI), especially in the domain of Generative Artificial Intelligence (Gen AI) and Machine Learning (ML), the architecture of Multi-agent systems (MASs) is again gaining popularity [15, 124]. That is due to its suitability for building complex, distributed systems where the autonomy of agents is essential for efficiently managing and hosting intelligent services and ML models, which often include integrated business logic [122]. Such systems may demand high scalability, particularly when serving a large volume of customers, where the performance output of these services might not be optimal. The application of MAS architecture, by its very nature, allows for the operation of multiple service instances and their effective coordination, thereby making the challenge of scalability more manageable [54]. Furthermore, these intelligent agents, which can be regarded as microservices, are typically designed for a single purpose. They contribute to a broader objective or goal in a limited capacity. In situations where the main goal needs a combined approach, linking together several agents that each have different jobs and putting their results together can be a good way to reach the desired goal [58].

Both of the described scenarios highlight the need for proper management of the coordination among interconnected agents, commonly referred to as orchestration. However, the implementation of agent coordination (be it sequential, parallel, conditional, or in other forms) in a MAS architecture, as evidenced by various research and projects, is not straightforward [127]. Each agent is an autonomous unit, requiring the establishment of its own communication mechanisms. This lack of standardized coordination often leads to inconsistencies in communication, as there is no assurance that the two agents intending to communicate have the appropriate communication channels in place. This issue becomes increasingly problematic in MAS that comprise a high number of agents, as the likelihood of communication failures increases with every new agent introduced into the

system [122].

A thorough analysis of literature in the domain of scientific papers, as well as industry standards, indicates that there is significant potential for advancement in the specification of agent communication flows and the coordination of agents within a MAS [24, 33]. Identifying this challenge led to the idea of developing a programming language that enables engineers to define agent relationships in the form of communication flows, along with a platform capable of interpreting and executing these specifications. Such a language and platform would streamline the process of establishing and managing communication channels between agents, thus improving the overall efficiency and effectiveness of MASs.

1.2 Introduction

Within the realm of Computer Science (CS), agents are software programs that execute tasks repeatedly to deliver resources or impact the requesting party, either directly or indirectly [122, 58]. A defining characteristic of an agent is its autonomy, enabling it to act as a standalone unit, designed to achieve goals without human direction or intervention. Agents follow both proactive and reactive behaviors. They perform tasks periodically according to predefined rules in a proactive manner and respond to events such as incoming messages in a reactive manner [54].

MASs comprise multiple such agents working together to deliver value beyond what individual agents can achieve. Due to their autonomy and ability to interact with other entities, agents can be also observed as heterogeneous microservices [122, 54]. In MAS, agents communicate to exchange information crucial for achieving their individual objectives, but which enables them to achieve higher-level, common goals. The proper orchestration of these agents is therefore essential to ensure the delivery of expected outcomes [58, 127].

The complexity of MAS increases with the number of agents, each adding a layer of complexity. One of the significant challenges in MAS is the specification of communication flows. Agents in MAS exchange messages via communication channels to inform each other about requests or their current status, making it crucial to specify these communication flows accurately for agent cohesion [59, 120].

With traditional approaches, each agent in a MAS is often designed to implement its own communication flows and business logic independently. However, this practice leads to redundancy in the codebase, reduced stability, lower readability, and limited extensibility when considering the MAS as a whole. This unit-oriented approach to agent design and implementation makes orchestrating agents within a MAS particularly challenging [59].

To address these challenges, a proposed solution involves constructing a programming language, based on process calculus, for specifying communication flows [105]. This

language, along with a declarative engine, facilitates the efficient processing of agent orchestration specifications and ensures coordinated execution based on those specifications. By using this programming language, engineers can define consistent communication flows between agents, enhancing the orchestration within the MAS. This method aims to streamline the communication process, reducing redundancy and increasing the scalability and maintainability of MAS.

In this research context, agent orchestration is viewed as a method for consistently specifying communication flows between pairs of agents, ensuring each agent is designed to communicate effectively with others. This approach is crucial for addressing the complexities and challenges inherent in MAS design and implementation.

To support the research, two literature domains have been analyzed with the goal of better understanding the problem domain, and identifying room for improvement: scientific papers and practices from the industry. These domains are recognized as the most relevant, as they cover approaches addressed from a scientific standpoint but also approaches from real-world use cases The databases that were searched for scientific papers are Scopus [27], WoS [21], and IEEE Xplore [43], while the papers that were analyzed are the ones with the highest citation count in the domain of communication flows in MAS.

The literature analysis in this study focuses on two areas: agent communication specification and agent orchestration. Agent communication specification is primarily examined through scientific papers, whereas agent orchestration is explored by reviewing the capabilities of available service orchestration providers, specifically in terms of coordinating agent communication.

Communication specification in MAS is a critical aspect that involves defining protocols for various elements such as message transport, headers, formats, flows, and other characteristics integral to effective agent interaction [81, 9]. Corkill [24] introduces the broadcasting concept in MAS, where a message is sent to all parties within a communication channel. This approach requires each agent to decide which messages are relevant and respond accordingly, filtering out the rest. Goldman and Ziberstein [33] developed an approach based on the broadcasting concept, taking into consideration that external agents (those not part of the observed MAS) may not be equipped to efficiently filter messages irrelevant to them. They propose restricting the number of agents within the MAS that communicate with external agents. In this setup, all agents within the MAS would direct their messages to one or several designated representative agents, who then relay these messages to the appropriate external agents. This approach to an extent resembles the holonic system concept [112, 89]. As a result of this proposal, one or more agents in the MAS are tasked with implementing sophisticated communication logic to accurately forward messages, thus centralizing the responsibility for defining communication flows on these agents. Further research dives into the development of sophisticated algorithms aimed at optimizing message delivery routes, ensuring the most efficient and

rapid communication among agents [3, 120, 17].

M. Berna-Koes et al. [13] address the challenge of transporting non-textual messages, such as video and audio content, within MAS frameworks designed for textual communication. They propose the innovative use of backchannels to facilitate the exchange of these media types. An example of this strategy could involve a single middleware agent in the MAS to which all other agents send their messages. This middleware agent then acts as a central hub for distributing various media types among the agents. In addition to message transport, the issue of message formats is critically examined by Luncean and Becheru [55], who advocate for the adoption of a common ontology and standardized message format. This approach aims to eliminate the uncertainties and incompatibilities associated with the use of custom message formats, thereby streamlining communication within the MAS.

The body of literature analyzed, encompassing studies [24, 33, 3, 17], addresses a wide range of message exchange concerns in MAS. However, these studies often remain confined to conventional design and implementation methods, lacking the application of novel and innovative approaches. As a result, they tend to offer solutions tailored to specific use cases rather than proposing a universal strategy that could be applied across various scenarios.

In contrast, the literature from industry practices focuses primarily on service orchestration providers. An agent in a MAS is often likened to a service that performs a specific task to add value. While each agent typically provides at least one service, not all services are considered agents due to the absence of autonomy [26]. In service-oriented architectures, especially in the context of microservices, the management of communication flows is referred to as orchestration [91]. The current market offers a limited number of service orchestration platforms, such as Kubernetes [18] and Docker Swarm [106], which predominantly concentrate on high-level orchestration tasks. These platforms are designed to manage the startup order of services and their clustering within the same environment, facilitating easy communication, efficient load balancing, and robust service deployment. However, they often overlook the intricacies of in-depth communication between individual services.

Kubernetes, for example, is an open-source platform that excels in managing services through declarative configurations, enabling the scaling of service clusters and automating their deployment. Its ecosystem is meticulously designed to define and monitor how services coexist, such as determining the clusters they belong to and the servers on which they are deployed. Despite these capabilities, Kubernetes does not address the detailed communication needs between services, leaving the responsibility of designing and implementing specific service features to the service developers themselves [90]. Similarly, Docker Swarm, operating as a mode within the Docker containerization platform, provides orchestration tools akin to those of Kubernetes but with a more lightweight approach to

infrastructure. Docker Swarm focuses on the combination of multiple clusters or swarms, yet, like Kubernetes, it does not manage the communication between the actual services [106, 85].

A range of other service orchestration providers, including Azure Container Instances [60], Google Cloud Run [36], and others, offer alternative solutions to Kubernetes and Docker Swarm. These providers emphasize the infrastructure required for service cohabitation, but do not specifically address service communication or collaboration. This focus on infrastructure over communication is understandable, given the diversity of communication requirements among different service types. Creating a universal solution for managing communication flows across all service types poses significant challenges due to the unique needs of each service [85].

Most service orchestration providers currently rely on configuration files to orchestrate service resources and dependencies. These configuration capabilities are primarily directed towards specifying service resources, such as memory and CPU allocation, as well as defining service dependencies within the orchestration environment. However, extending these capabilities to enable the specification of communication flows between services is a complex endeavor that is not yet feasible with the existing tools. The intricacies of communication in MAS, involving aspects like message prioritization, routing, and format compatibility, require more advanced and specialized solutions [18, 85].

Existing research and industry practices have significantly addressed agent communication and orchestration challenges in MAS. However, there remains a need for innovative solutions to manage the evolving complexities of these systems. Developing new methodologies, programming languages, and orchestration platforms is essential for efficiently handling the diverse and dynamic communication requirements in MAS, crucial for their advancement and scalability.

The literature review suggests significant potential for improving the way communication flows are specified within MASs. While scientific studies frequently explore design methodologies tailored to the unique parameters and constraints of MASs, they often stop short of offering practical tools or standardized frameworks for specifying agent interactions. In contrast, current service orchestration providers primarily focus on abstracting general service-level concerns, such as resource allocation, infrastructure management, and service dependencies. These approaches, while effective in traditional service architectures, fall short in addressing the dynamic, decentralized, and autonomous nature of agent-based systems.

1.3 Research questions

The primary research questions of this study focus on the construction of a programming language for agent orchestration within MAS architectures, along with an associated

declarative engine. This language aims to address the needs of contemporary domains and facilitate engineers' specification of communication flows. The specific research questions are as follows:

RQ1 What types of communication flows should be supported by the programming language considering the needs of the modern domains related to microservice architecture, artificial intelligence, and cloud computing?

RQ2 How to shape the syntax, semantics, and pragmatics of a programming language for the orchestration of heterogenous microservices in multi-agent system architectures by utilizing process calculus?

RQ3 How to support the design process of complex methods ensemble utilizing holonic systems?

1.4 Research objectives and hypothesis

The research objectives center on developing an artifact designed to simplify agent orchestration in MAS. This is achieved by utilizing a specialized programming language and a declarative engine. The specific objectives are as follows:

O1 Develop a programming language that enables the orchestration of heterogenous microservices in the multi-agent systems architecture, artificial intelligence, and cloud computing

O2 Develop a declarative engine based on process calculus that is capable of controlling communication flows between intelligent agents

The hypothesis being evaluated in this research is as follows:

H1 Programming language for communication flows specification based on process calculus shall enhance the orchestration of heterogenous microservices using multi-agent systems

1.5 Research paradigm, methods and requirements

The ultimate goal of the research is to develop an artifact that addresses previously identified drawbacks in specifying communication flows in MASs architecture. Therefore, pragmatism is chosen as a suitable research paradigm. Considering the artifact aims to impact how software engineers define communication flows and to what extent it influences how they implement the business logic (in the form of code readability, ease of integration,

etc.), a qualitative research approach is adopted. Design Science [115] is the methodology aligned with the desired outcomes and is therefore the one applied in this research.

To develop the artifact within this research, a set of requirements was collected that the artifact needed to satisfy. These requirements focus primarily on enhancing agent communication, while other intrinsic agent functionalities fall outside its scope. During the evaluation phase, the extent to which the requirements were met was examined, and the advantages and limitations of the artifact were assessed based on feedback from software engineers who evaluated it.

The primary methods used in the research include formalizing communication of agents in MAS using process calculus [105, 7], modeling the programming language based on predefined types of communication flows [14], and developing the programming language and declarative engine through software engineering methods [26].

For the formalization of agents' communication, process calculus is applied. This approach models concurrent systems by focusing on interaction, communication, and synchronization between agents. The key features shared by all approaches within process calculus include message-passing between independent processes, a small number of primitives and operators to describe processes, and specific algebraic rules for process operators. Reduction rules, an essential aspect of process calculus, enable the explicit description of communication, covering parallel composition, sequentialization, and transformation of input into output [105, 7]. Process calculus provides a formal foundation for modeling complex distributed systems in this research.

The modeling method aims to shape characteristics and elements of a new programming language using MAS communication development concepts [26] and holonic systems concepts [112, 89], within the boundaries defined by the formalization method [105, 7]. The implemented key concepts are oriented toward specifying communication flows, based on predefined requirements. Holons, as a concept for modeling complex distributed systems, allow for the organization of such systems into smaller units (holons) hierarchically [112, 89].

Software engineering methods were used for the development of the programming language and the declarative engine, based on the previously constructed model [66]. The outcome is a new programming language for constructing communication flow specifications and a declarative engine capable of processing these specifications.

1.6 Methodology

Design Science Research (DSR) is a research paradigm that emphasizes the creation and evaluation of practical solutions to complex problems through the development of innovative artifacts, and was therefore selected as the methodology for this research. DSR is particularly dominant in fields such as information systems, engineering, and CS, where

the development of functional and effective artifacts (for example tools, models, methods, and processes) is crucial. This paradigm is distinct from purely theoretical research as it aims to produce actionable knowledge that can be directly applied to real-world situations. The primary objective of this paradigm is to improve the state of the art and practice in a domain through the creation of artifacts that address unmet needs or improve existing solutions [115, 41].

This research paradigm indicates an iterative process, which involves the continuous refinement of the artifact based on systematic evaluation and feedback. This iterative approach includes phases: problem identification, objectives of solution, design and development, demonstration, evaluation, and communication. Each phase plays a critical role in ensuring that the artifact not only addresses the problem effectively but also contributes new knowledge to the field. The description of the phases is as follows [115]:

1. Problem identification: Recognizing and defining the specific problem that needs to be addressed.
2. Objectives of solution: Outlining what the artifact must achieve to address the problem effectively.
3. Design and development: Creating the artifact.
4. Demonstration: Showing how the artifact works in a practical scenario.
5. Evaluation: Assessing the artifact against criteria to validate its effectiveness and efficiency.
6. Communication: Sharing the results and knowledge gained with the broader community.

Evaluation is a critical component of DSR, ensuring that the artifact not only functions as intended but also contributes to the knowledge base of the discipline. This involves methods to assess the utility, quality, and efficacy of the artifact. Common evaluation methods include case studies, experiments, and field tests, which help to demonstrate the artifact's effectiveness in real-world settings [41].

By integrating both research methods and creative design principles, the DSR paradigm helps bridge the gap between theoretical exploration and practical application, particularly in fields like information systems, engineering, and management. Its emphasis on utility and innovation makes it especially relevant in rapidly evolving technological and business environments [41].

In the first phase of the chosen methodology, extensive literature analysis has been conducted in areas of MAS, microservice architecture, AI, orchestration, and cloud computing with a focus on the aspect of communication flows specification. Additionally,

this phase includes a comprehensive review of existing orchestration platforms, providing insights into their functionalities and best practices within the industry. The next step, Definition of objectives for a solution, was about gathering solution requirements and formulating a clear execution plan with specific objectives. Requirements are derived from the literature review, industry standards, and the project's research objectives, particularly within the context of the Orchestration of Hybrid Artificial Intelligence Methods for Computer Games (O HAI 4 Games) project [100]. This stage clarifies the communication features the programming language should implement, drawing upon concepts in MAS programming, holonic systems, and process calculus [112, 89, 105, 7]. In this phase, types of communication flows are defined that are supported by the language confined by limitations of agent development in MAS and requirements of holonic systems. A description of the application of process calculus for the communication flows specification was laid out by the completion of this phase, which is then used in the next phase, Design and development of artifacts.

In line with the execution plan, Design and development of artifacts phase involved constructing the programming language definitions that align with the identified requirements. It incorporates the previously described formalization, modeling, and software engineering methods. This includes designing the syntax, semantics, and pragmatics of the language, as well as implementing an interpreter based on a declarative engine capable of processing communication flows' specifications from the end-user, namely the programmer. Following the development phase, the artifact underwent demonstration by being applied to real-world MAS use cases. Specifically, use cases from the O HAI 4 Games project, which deal with various aspects of A Massively Multiplayer Online Role-playing Games (MMORPGs), cognitive agents in gamification, serious games, and holographic games [100], are utilized for testing and validation.

The subsequent phase involves the evaluation of the solution. The artifact was assessed using the Framework for Evaluation in Design Science research (FEDS) framework [116] to determine its effectiveness in equipping a programming language with features for managing communication flows. This evaluation follows four steps: defining goals, selecting strategies, determining evaluation properties, and designing evaluation episodes. Feedback will be collected from the O HAI 4 Games project team, as well as from independent experts with experience in cloud systems, AI, and microservices.

The evaluation combines formative evaluations in controlled environments and summative evaluations in real-world scenarios. Formative evaluations focus on iterative testing to resolve technical issues, while summative evaluations assess integration and efficacy through practical use cases [116]. Key properties evaluated include artifact quality, interoperability, modularity, usability, performance efficiency, and compliance with programming language standards. Feedback will be analyzed qualitatively using thematic analysis, and results will be documented in a report summarizing strengths, challenges,

and recommendations for improvement. This structured approach ensures a comprehensive evaluation of the artifact's technical and functional capabilities [84, 111, 114, 62].

Finally, the results and findings, including the problem, the artifact, its utility, and effectiveness, are communicated to the wider research community. The evaluation outcomes are presented to the scientific audience through research papers and discussions. Additionally, any identified areas for improvement are addressed, and the hypothesis is validated based on the evaluation feedback provided by the evaluators, as noted previously.

The expected scientific contributions of this research are diverse and substantial. Firstly, the development of a programming language specifically designed for the specification of communication flows in MAS stands as a primary contribution. This language is anticipated to enhance agent orchestration by providing a framework for consistent communication flows specification across all agents. One of its key features is the ability to offer a higher degree of extensibility or modularity in communication flows, which is crucial for adapting to various scenarios within MAS. Additionally, the language will support hierarchical structures within MAS, facilitating more organized and efficient agent interactions.

The second major contribution is the creation of a declarative engine. This engine will be capable of processing communication flow specifications defined using the new programming language. A key feature of this engine is that it shifts the responsibility for implementing communication logic away from software engineers, allowing them to focus on defining high-level specifications rather than low-level implementation details. As a result, it provides a flexible and user-friendly platform that supports the customization of communication flows according to specific requirements and scenarios within MAS. This approach significantly enhances both the efficiency and maintainability of communication in MAS, making the engine a valuable tool for software engineers working in this domain.

The expected social contribution of this research is practical and community-oriented, as it involves providing an open-source solution to both the scientific and professional communities. This contribution is grounded in the idea of making the research outputs, such as the programming language for MAS communication flows and the declarative engine, readily available for usage and evaluation.

The research is structured into nine chapters, which map to the phases of the DSR methodology [41]. Chapter 1 presents the motivation for conducting the research, a description of the relevant domains and fields, and a review of related scientific and professional literature. This is followed by chapter 2, which describes key concepts, definitions, architectures, and associated tools, with the aim of better understanding the research foundations. In chapter 3, an analysis of existing development frameworks and libraries for MAS development is conducted. Chapter 4 outlines a set of requirements for the development of the artifact, whose implementation is elaborated in chapter 5.

Chapter 6 demonstrates the implementation of the artifact through use cases within the O HAI 4 Games project. Furthermore, Chapter 7 addresses the evaluation conducted on the artifact. Finally, Chapters 8 and 9 provide insights into possible improvements and final conclusions.

Chapter 2

Related work

This chapter presents a detailed explanation of the services, tools, definitions, and concepts that support the technology behind the artifact.

2.1 Agentic concepts, definitions, and architectures

The upcoming sections dive into the foundational aspects of agentic systems. They begin by defining what an agent is and explore the key characteristics that distinguish agents from other computational entities. Common architectures that support agent design are then examined, highlighting how these frameworks enable complex behaviors and interactions. The discussion also covers the relationships and dynamics between multiple agents, emphasizing their capacity for collaboration, competition, and co-evolution.

2.1.1 Agent

Within the field of Computer Science (CS), an agent is a software entity that performs tasks autonomously, often in response to changes in its environment or at scheduled intervals. Agents are designed to operate without continuous direct human oversight, guided by their built-in logic and predefined conditions. This autonomy is central to the concept of an agent, distinguishing it from simple scripts or programs that require manual initiation. An example task that an agent might perform is to periodically collect weather data from an external source and send an email message to a group of users who have an interest in it [58].

Key characteristics of agents are autonomy, reactivity, proactivity, and social ability. Autonomy indicates that an agent has the capability to make decisions and act on them without human intervention. This decision-making process is typically guided by a set of rules or an algorithm that helps the agent determine the best course of action based on its objectives and the information available to it. Agents are reactive, meaning they can perceive their environment and respond to changes within it in a timely manner.

For example, an agent might monitor stock prices and execute trades based on specific market conditions [67]. Additionally, agents can be proactive, taking initiative based on their goals rather than just reacting to the environment. These capabilities allow agents to function effectively in dynamic settings, adapting their behavior as needed to achieve their designated tasks. While not always required, many agents possess the ability to interact with other agents or systems in a meaningful way to complete tasks. This interaction can be as simple as sending data to another system or as complex as negotiating task execution with other agents. This social ability is particularly important in systems where multiple agents must work together to achieve a common goal. A cluster of such connected agents is called Multi-agent system (MAS) [122, 58].

An agent can be formally defined by the following components [92, 35]:

- Percepts (P): Sensory inputs from the environment.
- Actions (A): Possible outputs or behaviors.
- State (S): Internal representation of the agent's knowledge or beliefs.
- Transition Function ($t : S \times P \to S$): Updates the agent's state based on percepts.
- Action Function ($f : S \to A$): Determines actions based on the current state.

A rational agent's behavior can be captured by the agent function which maps percept sequences to actions [35]:

$$f : P^* \to A$$

Behavioral properties are described as follows [92]:

- Reactivity:
$$\forall p \in P, \exists a \in A \text{ such that } a \text{ responds to } p$$
- Proactivity: Goal-driven action selection via utility functions or planning.
- Autonomy: Decisions independent of external control, formalized as:
$$\nexists f' \neq f \text{ overriding } a$$

Designing effective agents requires careful consideration of the agent's operational environment and tasks. Agents must be equipped with the appropriate sensors to perceive their environment accurately and actuators to perform actions. Moreover, the decision-making mechanisms need to be robust enough to handle the complexities of the real-world, where unexpected events can occur. As agents operate autonomously, there are significant ethical and security considerations. Ensuring that agents do not perform unauthorized

actions or expose sensitive information is crucial. Figure 2.1 previews the architecture of a simple agent [30].

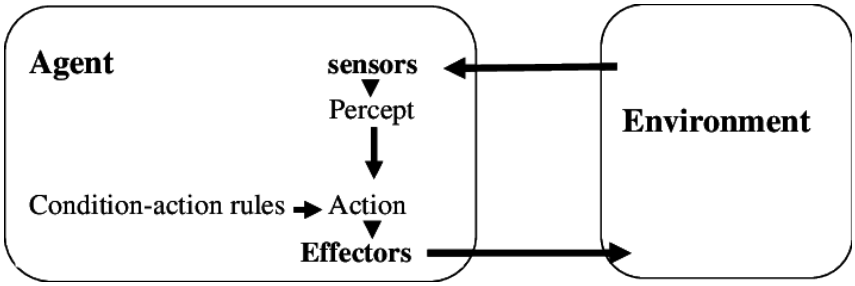

Figure 2.1: Architecture of a simple agent [44]

2.1.2 Intelligent agents

An intelligent agent is deemed intelligent due to its ability to perceive its environment, process information, and make decisions that guide its actions toward achieving specific goals. Its intelligence comes from several core attributes: the capacity to learn and adapt from past experiences through Machine Learning (ML) and Deep Learning (DL) techniques, enabling improved decision-making. It has the ability to autonomously select optimal actions using optimization strategies, utility functions, or rule-based logic. It uses sensing abilities like Computer Vision (CV) and Natural Language Processing (NLP) to interpret sensory data and understand context. It shows adaptability and flexibility in dynamic environments through Generative Artificial Intelligence (Gen AI) models that anticipate scenarios and modify strategies accordingly. Finally, it has the autonomy to execute tasks independently, self-managing priorities, and choosing actions that best align with its objective [123]. These characteristics empower intelligent agents to act effectively in complex, changing environments, providing robust solutions in various fields [67].

Recent advancements in the field of Gen AI have seen a significant increase in the traction of agents, particularly as they become more integrated and capable within various sectors [16]. This trend is driven by the continuous evolution of Artificial Intelligence (AI) technologies that enhance the autonomy and decision-making capabilities of agents, making them increasingly valuable in complex, real-world applications. Agents in Gen AI are being developed to exhibit higher levels of intelligence and autonomy, moving closer to achieving human-like cognitive abilities [71]. These agents are designed to perform a wide range of tasks that require understanding, learning, and decision-making capabilities that were traditionally thought to be exclusive to humans [83].

2.1.3 MASs

MASs are composed of multiple interacting agents, each with their own set of capabilities, objectives, and knowledge. Agents in a MAS can work collaboratively or competitively,

depending on the system's overall objective, to solve problems that are beyond the capabilities of individual agents. They communicate the results of their processing by exchanging messages [122, 58]. An example of a MAS use case might involve a scenario with three sequentially communicating agents. The first agent gathers weather data (temperature, humidity, etc.) from an external source and passes it on to the second agent. This second agent is tasked with forecasting the weather for the next 6-12 hours. After computing the forecast, it relays the results to the third agent, which then sends the information to the interested audience.

The architecture of a MAS defines how agents are organized and how they interact within the system. There are several types of architectures, each with its own advantages and applications [94, 25]:

- Centralized: In this setup, a central agent has control over the other agents, making decisions on their behalf. This architecture simplifies decision-making but can become a bottleneck and a single point of failure.

- Hierarchical: Agents are organized in a hierarchy, where some agents have authority over others. This allows for scalable decision-making but can suffer from inefficiency due to the overhead of communication and control.

- Distributed: All agents operate at the same level of authority and coordinate their actions among themselves. This architecture promotes scalability and robustness but requires sophisticated coordination mechanisms.

- Heterarchical: Similar to distributed architectures, but with a more flexible structure that allows for dynamic reconfiguration of relationships among agents. This is particularly useful in environments where the tasks and roles of agents frequently change.

- Holonic system: the concept of holons, which are entities that are both wholes and parts of other wholes [25].

Below is the formalization of a MAS, describing its structure, execution, and evolution [121].

- Dynamical State:

$$D = (\sigma, \gamma)$$

  where:

  - σ is the environment state
  - γ is the set of consumptions (effects reserved for agents)

- Actors:
$$A = \{a_1, a_2, \ldots, a_n\}$$
where A is the set of agents.

- Ongoing Activities:
$$OA = \{oa_1, oa_2, \ldots, oa_m\}$$
where OA is the set of ongoing environment activities (e.g., moving objects, evaporating pheromones).

- Influences:
$$I = I_{\text{agents}} \cup I_{\text{ongoing}}$$
where:
  - I_{agents} are influences produced by agents
  - I_{ongoing} are influences produced by ongoing activities

- Regional Synchronization:
$$R = \{R_1, R_2, \ldots, R_k\}$$
where each R_i is a group of agents that synchronize locally. Different regions act asynchronously.

- Execution Functions:
$$\begin{aligned} \text{Exec} &: (\sigma, \gamma) \to I_{\text{agents}} \\ \text{Exec\_OA} &: (\sigma) \to I_{\text{ongoing}} \end{aligned}$$

- Reaction Function:
$$\text{React} : (\sigma, I) \to (\sigma', \gamma')$$
where the environment reacts to the combined influences.

- Evolution Cycle:
$$\text{Evol}(D) = \text{React}(\text{Exec}(D))$$
expanded as:
$$\begin{aligned} D &= (\sigma, \gamma) \\ I_{\text{agents}} &= \text{Exec}(\sigma, \gamma) \\ I_{\text{ongoing}} &= \text{Exec\_OA}(\sigma) \\ I &= I_{\text{agents}} \cup I_{\text{ongoing}} \\ (\sigma', \gamma') &= \text{React}(\sigma, I) \\ D' &= (\sigma', \gamma') \end{aligned}$$

In addition to the architecture, which largely dictates the relationships between agents, another critical factor influencing the design and implementation of a MAS is whether agents are distributed across multiple servers. If agents are spread across different servers, it introduces risks associated with performance, security, and data consistency. Communication latency and synchronization issues might arise, requiring robust strategies to maintain system coherence and reliability [127]. This also affects the choice of development tools, as not all frameworks offer seamless support for managing agents distributed across multiple servers. Selecting the appropriate framework becomes essential to ensure efficient inter-agent communication, resource sharing, and system resilience, while minimizing risks related to data breaches, unauthorized access, and degraded performance [58, 54]. Figure 2.2 outlines the general structure of a MAS.

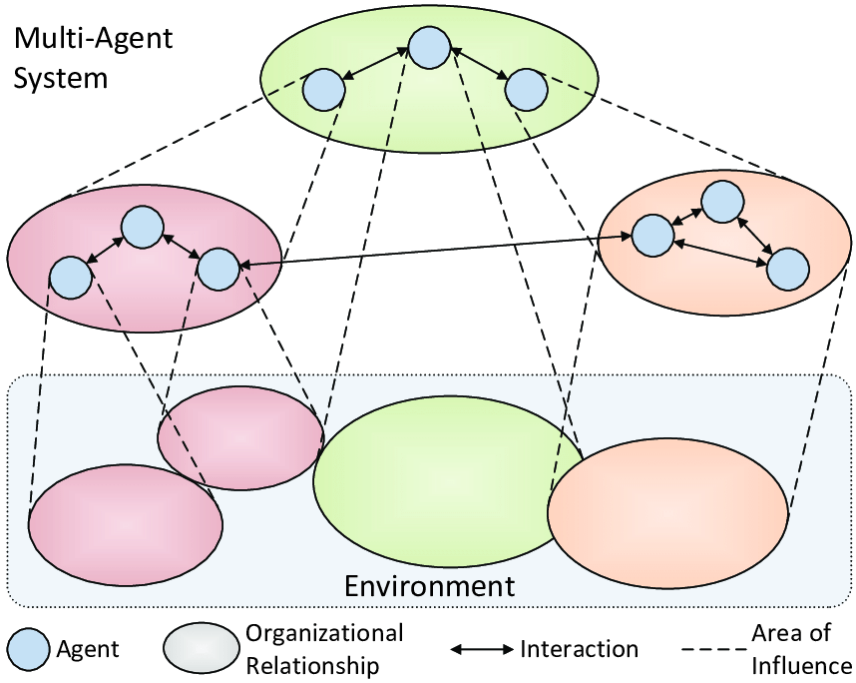

Figure 2.2: General structure of MAS [51]

An agent within a MAS is subject to specific design and implementation in order to embody characteristics of independent functioning. Some of these characteristics are related to the communication flows specification, availability of the needed resources, and so on. Because agents are autonomous units, it is possible they use different technologies and approaches for implementing the business logic, including communication flows. As a consequence, each agent individually may be required to implement communication flows, which leads to several challenges [127]:

- Each agent individually implements communication flows, leading to code redundancy and increasing the potential for failure
- Inconsistencies in communication flows (for example, agent A is designed to com-

municate with agent B, but agent B is not designed to communicate with agent A)

- Weak extensibility (adding a new agent to the MAS requires changes to the existing agents)
- Lack of a single source of truth to describe communication flows in the MAS
- Potential conflicts in retrieving shared resources (e.g. race conditions)

Several popular frameworks are available for developing and supporting MASs, including Microsoft AutoGen [6], LangGraph [50], Smart Python Agent Development Environment (SPADE) [108], CrewAI [82], among many others.

2.1.4 Holonic systems

Holonic systems in MAS represent a sophisticated organizational structure where individual entities, known as holons, can act both autonomously and cooperatively. In this framework, holons embody both a part and a whole simultaneously, functioning as independent agents that contribute to larger organizational units. This dual nature allows them to possess their goals, resources, and decision-making capabilities while coordinating with other holons to achieve higher-order objectives. Holonic systems are especially advantageous in dynamic and complex environments, as they offer flexibility, scalability, and robustness. These properties make them ideal for applications in manufacturing, logistics, and robotics, where adaptability and efficient resource management are crucial. Their hierarchical yet decentralized organization enables them to effectively tackle complex tasks through distributed problem-solving, by delegating functions across different levels of the hierarchy, leading to more efficient and responsive systems [112, 89].

In the realm of MAS, holonic systems introduce a structured and recursive way of organizing agents, known as holons, into a cohesive system. A holonic agent is a self-similar entity composed of holons as substructures, where the compound holon is qualified as a super-holon, and the holons that compose a super-holon are called subholons or holon members. This recursive architecture allows for complex systems to be broken down into manageable, semi-autonomous units (holons) that can efficiently achieve a common goal through collaboration and self-organization [31].

The following provides a formalisation of holonic systems, detailing their core components and relationships through graph-theoretic definitions, based on prior work [96].

An organizational unit (OU) is defined as:

- Any agent a is an organizational unit.

- More generally, an organizational unit is a labeled graph:

$$OU = (O, R, C)$$

  where:

  - O is a set of organizational units (nodes),
  - R is a labeled set of roles (edges/arcs),
  - C is a criteria of organizing (e.g., objective, function, role).

An organizational process (OP) is defined as:

- Any atomic activity p performed by an individual agent is an organizational process.
- More generally, an organizational process is a labeled directed graph:

$$OP = (P, R, C)$$

  where:

  - P is a set of processes (nodes),
  - R is a set of ordered relations between processes (edges),
  - C is a criteria of organizing (e.g., resource dependency, sequential execution).

An organizational strategy (OS) is defined as:

- Any measurable objective s achievable by an atomic activity is a strategy.
- More generally, an organizational strategy is a labeled directed graph:

$$OS = (S, R, C)$$

  where:

  - S is a set of strategies (nodes),
  - R is a set of relations between strategies (edges),
  - C is a criteria of connection (e.g., influence, responsibility).

An organizational knowledge artifact (OK is defined as:

- Any accessible knowledge artifact k (such as data, rules, protocols) is an organizational knowledge artifact.

- More generally, organizational knowledge is a labeled graph:

$$OK = (K, R, C)$$

  where:

  - K is a set of knowledge artifacts (nodes),
  - R is a set of relations between artifacts (edges),
  - C is a criteria of organizing knowledge (e.g., thematic grouping, shared accessibility).

A critical aspect of implementing holonic systems in MAS is the concept of the mediator agent. The mediator acts as the interface between the agents within a holon and those outside it, facilitating communication and coordination. It serves two main functions: representing the holon to external entities and brokering or supervising interactions among the sub-holons. This dual role of the mediator is essential for maintaining the integrity and functionality of the holonic system, ensuring that each holon can effectively contribute to the system's overall objectives while retaining its autonomy [31]. Figure 2.3 previews a structure of MAS in holonic system settings.

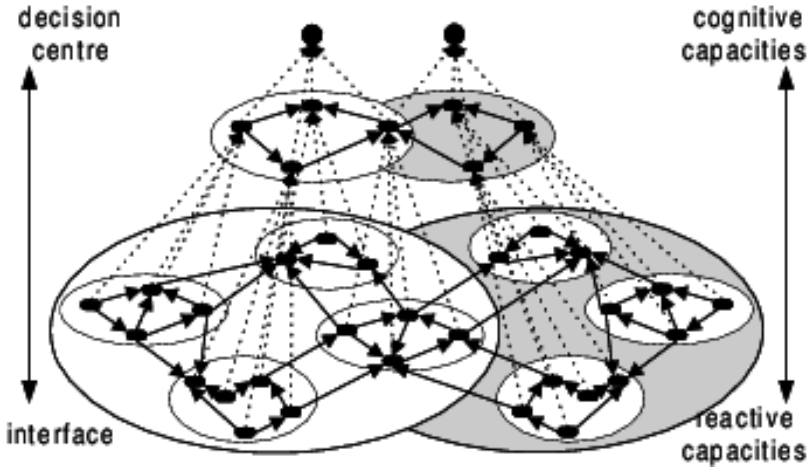

Figure 2.3: General structure of holonic system [1]

2.2 Microservices and orchestration

These sections explore the dynamic and modular world of microservices architectures, a design approach that structures applications as collections of loosely coupled services. We will begin by defining microservices and discussing their core principles, including their independence in deployment and scalability. Further research focuses on the orchestration of these services, examining how various orchestration tools and platforms manage interactions and dependencies between services efficiently.

2.2.1 Microservices

Microservice architecture is an architectural style for designing and developing software systems where each system is designed to serve a single, specific purpose. This approach contrasts with monolithic architecture, where a single system might provide multiple services and be used for various purposes. In microservice architecture, services are loosely coupled, meaning they communicate with each other through various communication channels and APIs. The loose coupling of these services allows for independent deployment, enabling only the services that have been updated or changed to be deployed. This is a significant advantage over monolithic architecture, where the entire system often needs to be redeployed when any single part is updated [113].

Services that adhere to microservice patterns are commonly referred to as microservices. These microservices are highly autonomous, meaning they are sophisticated enough in their business logic to deliver outputs independently, without sharing code with other services. When necessary, microservices communicate with other microservices to deliver end-to-end functionality that they cannot provide alone. Importantly, microservices are specialized in their functions and are designed not to duplicate the services offered by other microservices within their cluster. For example, a microservices cluster might include an authentication microservice, a microservice that offers a public API for user interaction, and another microservice that performs specific business logic. In a monolithic architecture, all these services might exist within the same system, but this can make deployment more challenging [10].

The autonomy and specialization of microservices, as well as their ability to perform tasks independently, allow them to be interchangeably referred to as agents. Each microservice or agent is focused on providing a distinct type of service, contributing to the overall functionality of the software system in a modular and flexible manner [10, 26].

2.2.2 Containerization and software systems orchestration providers

Contemporary software systems are increasingly complex, a complexity that can be analyzed from various aspects, including infrastructure setup, technology, resource requirements, and the need for scalability, among others. From the infrastructure perspective, this complexity often relates to the need for adequate infrastructural support. Such support is essential for interconnected systems to communicate effectively, share the same network and resources, and perform similar functions. On the technological front, the constant emergence of new, more efficient, and advanced technologies presents challenges in adapting software systems for successful development and deployment. Additionally, resource allocation is a critical aspect, involving the setup of necessary resources like memory storage, CPU, GPU, and process allocations, to ensure the system functions as

expected [20, 26].

Another key aspect of complexity is the need for scalability, which involves dynamically allocating resources based on traffic volume in specific situations. This is crucial for delivering high-throughput applications. These complexities are just a few of the aspects needed for software systems to be stable and perform optimally within production environments. However, addressing these challenges to ensure the functioning of systems would be difficult without proper technological support. This is where the roles of system containerization and software systems orchestration providers become crucial. They offer solutions to manage these complexities, enabling more efficient and resilient software systems [127].

Containerization of a software system involves packaging the software into an isolated environment equipped with all necessary resources. This process allows the software to function both as a standalone unit and in cohesion with other systems. One of the key benefits of containerization is its ability to ensure that containerized software performs consistently, regardless of the environment it is deployed in, such as different operating systems. This consistency ensures that systems developed and tested in a local environment will function similarly in a production environment, making them more predictable and controllable [26].

Additionally, containerization supports several critical aspects of modern software development, including portability, scalability, fault tolerance, and agility. Portability allows for easy movement of software across different environments, while scalability ensures that resources can be dynamically adjusted to meet varying demands. Fault tolerance enhances system reliability by isolating issues within individual containers, preventing widespread system failures. Lastly, agility is achieved through the ease of updating and deploying containers, which supports rapid development cycles and quick responses to changing requirements [20].

Containerization addresses the challenge of preparing systems for deployment but does not encompass the actual deployment process. This is where software systems orchestration providers play a crucial role. In simple terms, these providers are designed to deploy containerized systems into a production environment. Essentially, they take the containerized systems and place them in an environment according to a defined deployment specification. This specification includes details such as which containerized systems should be deployed, the number of instances to deploy, scalability rules, and whether systems within the same deployment should be visible and accessible to each other, among other factors [91].

This coordination of systems can be described as systems orchestration, as it involves careful coordination, or orchestration, of the systems to ensure they work cohesively within their cluster. Orchestration is critical for the smooth operation and scalability of the system as a whole. Some well-known service orchestration providers include Docker

Swarm [56], Kubernetes [18], Azure Container Instance [60], and Google Cloud Run [36], among others. Each of these platforms offers unique features and capabilities to manage and deploy containerized systems effectively [18, 106, 91].

2.3 Context free grammars

In CS, grammars play a critical role in the design and analysis of both programming languages and compilers. A grammar in this context is a formal set of rules that defines the syntactic structure of a programming language. This set of rules helps determine whether a sequence of characters conforms to the language syntax and is thus interpretable by the compiler. Grammars are fundamental for the compiler's ability to interpret and process high-level programming instructions into a form that can be executed by a computer [103, 107].

The compilation process initiates with lexical analysis, where source code is broken down into fundamental elements or tokens such as keywords, operators, identifiers, and literals, simplifying raw input for further compilation phases. This is followed by syntax analysis, which checks syntactic correctness against grammatical rules, often defined by a Context-free grammar (CFG), and builds a parse tree to structurally represent the inputs in alignment with the language's grammar. After verifying syntax, the compiler proceeds to semantic analysis, ensuring that the parse tree adheres to language semantics, including type correctness and scoping rules, crucial for identifying errors that, though syntactically correct, may be semantically inappropriate or illegal. Subsequently, intermediate code generation converts the source into an intermediate language, which is a lower-level standardized code that is easier to optimize and translate into the target machine language. The optimization phase then enhances the performance of this intermediate code, potentially reducing code lines, optimizing variable handling, and improving resource allocation. The final phase, code generation, translates the optimized code into the target platform's machine language, culminating in executable code. This comprehensive process highlights the essential role of grammars in transforming high-level programming languages into precise, executable machine instructions, bridging human-readable code and machine operations [107, 61].

CFGs are a class of grammars that are especially crucial in both linguistics and CS for defining the syntax of languages in a way that is independent of context. A CFG consists of a set of production rules that describe how terminal symbols (basic units of the language) and nonterminal symbols (combinators of terminals and other nonterminals) can be combined to generate strings. This formalism allows CFGs to describe the syntax of a wide array of programming languages and many natural languages with a straightforward yet powerful set of rules. In CS, CFGs are widely employed in the design of compilers and interpreters, where they are used to generate parsers that help convert

high-level code into a structured format (like parse trees) understandable by machines. The ability of CFGs to encapsulate the necessary syntactical rules without reference to semantic content makes them highly effective for analyzing the correctness of syntax and structuring complex programming and linguistic constructs [86, 38].

A CFG is defined as a 4-tuple [86]:

$$G = (V, \Sigma, P, S)$$

where:

- V (Nonterminal symbols):
  A finite set of nonterminal symbols, which serve as placeholders for patterns in the language. These symbols are used to define the structure of strings and are eventually replaced by terminal symbols during derivations.

- Σ (Terminal symbols):
  A finite set of terminal symbols, disjoint from V, which constitute the actual characters or tokens of the language. These are the basic symbols that appear in the strings of the language generated by the grammar.

  $$\Sigma \cap V = \emptyset$$

- P (Production rules):
  A finite set of rules of the form:

  $$A \to \alpha$$

  where:

  - $A \in V$ (a single nonterminal symbol),
  - $\alpha \in (V \cup \Sigma)^*$ (a sequence of terminal and/or nonterminal symbols, including the empty string ε).

- S (Start symbol):
  A distinguished symbol $S \in V$ from which derivations begin. The grammar generates strings by applying production rules starting from S.

2.4 Process calculus

Process calculus which also goes by process algebra is a collection of approaches in CS used to model concurrent systems. These systems are characterized by the execution of multiple processes in parallel, which may interact or communicate with each other. Process calculus

provides a formal framework for describing these interactions and analyzing the properties of concurrent systems [12, 7].

The fundamental elements of process calculus include message-passing mechanisms that enable communication between independent processes, a concise set of primitives, and operators designed to describe and compose processes. A key aspect of process calculus is its algebraic and reduction rules, which facilitate clear descriptions of interactions like parallel composition, sequentialization, and transforming inputs into outputs. These rules provide a formal framework for defining and analyzing process behaviors. By offering a structured approach to model complex distributed systems, process calculus serves as an essential theoretical foundation, making it indispensable for the research at hand [105, 79].

Several types of process calculus have been developed, each with unique features and specific areas of application [12]:

- Calculus of Communicating Systems (CCS): This type is focused on the communication between processes through labeled actions and their corresponding co-actions, making it effective for modeling systems where interaction patterns are a key concern.

- Communicating Sequential Processes (CSP): CSP models processes that interact via synchronized events, providing a robust framework for describing systems where coordination and synchronization are critical.

- π-Calculus: This calculus extends the capabilities of CCS by allowing the communication structure itself to change dynamically. It is particularly useful for modeling mobile systems where the configuration of the system can evolve over time, reflecting changes in connectivity and communication channels [76].

- Ambient Calculus: This calculus emphasizes the movement of processes in and out of bounded contexts, known as ambients. It is particularly suited for modeling mobile computing scenarios where computational entities need to move between different environments, encapsulating mobility and dynamic context changes.

- Join-Calculus: This variant simplifies and unifies aspects of the π-Calculus, especially in handling communication channels. It streamlines the implementation in programming languages, making it more practical for developing distributed applications.

Process calculus is widely used in the design and analysis of computer systems where concurrent operations are prevalent. This includes distributed systems, network protocols, and multi-core processors, where ensuring the correct sequencing and interaction of concurrent processes is critical. Process calculus helps in verifying properties such as

deadlock-freedom (a system state where no progress is possible) and liveliness (ensuring that certain actions will eventually occur). Additionally, it has been applied in the development of formal verification tools that can automatically check whether a system meets specified requirements. These tools are invaluable in domains where system failure can have serious consequences, such as in aerospace, automotive, and critical infrastructure systems [8, 12].

The formal syntax is outlined below, based on [28, 8, 12]:

- Processes: Processes are denoted as P, Q, R, representing concurrent or sequential computations.
- Basic syntax:
  - 0: Represents the *inactive process*.
  - $a.P$: Denotes an action a followed by process P.
  - $P + Q$: Represents a *choice* between processes P and Q.
  - $P \mid Q$: Denotes the *parallel composition* of processes P and Q.
  - $\tau.P$: Represents a *silent action* followed by process P.
  - $[x = y]P$: Conditional execution of P if $x = y$.
  - $A(x)$: A process defined by a recursive equation.
- Channels and communication:
  - $a(x).P$: Receive a message x on channel a and continue as P.
  - $\overline{a}\langle x\rangle.P$: Send a message x on channel a and continue as P.

Following are operators in Process Calculus:

- Parallel composition ($|$): $P \mid Q$ denotes concurrent execution of P and Q.
- Choice ($+$): $P + Q$ denotes non-deterministic branching; either P or Q proceeds.
- Sequential composition: $a.P$ denotes action a, followed by process P.
- Restriction (νa): Restricts the scope of a channel a, making it private to a process.
- Replication ($!P$): Represents infinite repetition of process P.

The semantics of process calculus is defined using transition systems and structural operational semantics.

- Communication:

$$a.P \mid \overline{a}\langle v\rangle.Q \rightarrow P[v/x] \mid Q$$

- Choice:

$$a.P + b.Q \xrightarrow{a} P$$

- Parallel composition:

$$P \xrightarrow{\alpha} P' \implies P \mid Q \xrightarrow{\alpha} P' \mid Q$$

- Restriction:

$$P \xrightarrow{\alpha} P' \implies \nu a.P \xrightarrow{\alpha} \nu a.P'$$

- Silent actions:

$$\tau.P \xrightarrow{\tau} P$$

Following are algebraic laws that describe the formalism:

- Commutativity:

$$P \mid Q \equiv Q \mid P$$

- Associativity:

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

- Identity:

$$P \mid 0 \equiv P$$

- Distributivity:

$$P \mid (Q + R) \equiv (P \mid Q) + (P \mid R)$$

- Expansion:

$$(a.P \mid b.Q) \equiv a.(P \mid b.Q) + b.(a.P \mid Q)$$

2.4.1 π-Calculus

π-Calculus is a mathematical model used to describe and analyze the behaviors of concurrent systems. It incorporates dynamic topology changes in network structures. This flexibility makes π-Calculus particularly effective at modeling complex systems where the communication structure can evolve during execution [76].

π-Calculus revolves around the concept of naming, where channels for communication are represented by names. These names can be dynamically created, communicated, and modified, allowing for the representation of mobile systems where the configuration of connections between processes can change. This is a significant enhancement over static models, providing a robust framework for modeling systems such as mobile networks, distributed systems, and even biological processes [76].

The key elements of this model are as follows [76]:

- Names and channels: Names serve as channels for communication between processes. They are essential for sending and receiving messages, allowing processes to interact.
- Prefixes: These represent the basic actions in π-Calculus, including output, input, and silent actions. Each prefix specifies a type of action and the process that follows after the action is performed.
- Agents: Agents (or processes) are the core entities in π-Calculus that perform actions. They are composed using various operators, such as parallel composition, summation, and restriction, to build more complex behaviors from simpler ones.

Processes can send and receive channel names, effectively altering the network's structure as the computation proceeds. This ability to communicate names means that processes can dynamically reconfigure the communication flows. For instance, a process can send a channel name to another process, enabling the recipient to communicate with a new set of processes through the received channel. This dynamic aspect makes π-Calculus particularly suitable for representing scenarios in distributed computing, where processes may need to establish, modify, or terminate connections on-the-fly [76].

π-Calculus has found applications in various fields due to its expressive power in modeling concurrency and mobility. In CS, it is used to analyze and verify the behavior of concurrent systems, ensuring properties such as deadlock freedom and liveness. Its principles are also applied in the design of programming languages and verification tools for concurrent and distributed systems [76].

Following are the key components of π-Calculus [76]:

- Names (N): A potentially unlimited set of names, N, denoted by {a,b,c,. . . ,z,a1,b1,. . . } used for channels, variables, and values.
- Identifiers (A): A set of identifiers representing processes or agents, defined by the expressions.

The expressions and the associated semantics are as follows [76]:

- Prefixes:
  - Output ($\overline{a}x$):
    Sends the name x (data) over the channel a and continues as process P.
    Notation: $\overline{a}x.P$
  - Input ($a(x)$):
    Receives a name (data) over the channel a, stores it in variable x, and continues as process P.
    Notation: $a(x).P$

  - Silent/Inactive (τ):
    Performs an internal action (silent move) and continues as process P, independent of external interaction.
    Notation: $\tau.P$

- Agents:
  - Nil (0):
    Represents the inactive process that cannot perform any actions.
    Notation: 0
  - Prefix ($\alpha.P$):
    Represents a prefix action α followed by process P.
    Notation: $\alpha.P$
  - Sum ($P + Q$):
    Non-deterministic choice between processes P and Q.
    Notation: $P + Q$
  - Parallel composition ($P \mid Q$):
    Processes P and Q run in parallel and can communicate via shared channels.
    Notation: $P \mid Q$
  - Match ($(x = y)?P$):
    If names x and y are equal, proceed as process P. Otherwise, no action.
    Notation: $(x = y)?P$
  - Mismatch ($(x \neq y)?P$):
    If names x and y are different, proceed as process P. Otherwise, no action.
    Notation: $(x \neq y)?P$
  - Restriction ($(\nu x)P$):
    Restricts the scope of the name x to process P. The name x becomes private.
    Notation: $(\nu x)P$
  - Agent identifier ($A(y_1, \ldots, y_n)$):
    Represents a process call to an agent A with parameters y_1 to y_n.
    Notation: $A(y_1, \ldots, y_n)$

- Definitions:
  - Each agent identifier $A(x_1, \ldots, x_n)$ is defined as a process P with the condition that if $i \neq j$, then $x_i \neq x_j$ (i.e., no parameter name repetition).

The above formalization defines the fundamental syntactic constructs of the π-Calculus. Names (N) are drawn from an infinite set and serve multiple roles, functioning as communication channels, variables for received values, and the transmitted values themselves.

Identifiers (A) represent agent names, providing a mechanism for defining recursive processes. Processes are constructed through a combination of simple operations: action prefixing, nondeterministic choice, parallel composition, conditional matching and mismatching of names, name restriction to control the scope of private names, and invocation of agent definitions [76].

Prefix actions allow processes to interact with their environment or progress internally, capturing the fundamental notions of sending and receiving messages or performing silent moves. The operators for choice and parallel composition provide the basis for modeling nondeterministic behavior and concurrent execution, respectively. Matching and mismatching enable conditional branching based on name equality, while restriction ensures that names can be localized to particular process scopes, supporting encapsulation. Together, these constructs form the basic language for describing mobile and communicating processes, setting the stage for specifying operational semantics that govern how processes evolve through interaction [76].

2.4.1.1 Pict

Pict is a programming language designed around the principles of the π-Calculus. It leverages the π-Calculus's ability to handle dynamic communication structures, making it suitable for modeling and implementing concurrent and distributed systems. The language relies primarily on a concept of a channel, as the primary way of communication, which aligns with π-Calculus's emphasis on the dynamic interaction topologies. This approach allows Pict to support a wide range of high-level constructs, including data structures, higher-order functions, and concurrent control structures, all through the paradigm of message passing [80].

Pict's core language is a version π-Calculus comprised of explicit types, extended with practical features such as records and pattern matching. This foundation simplifies the compilation of Pict programs and allows for various optimizations common to functional languages. The design goals of Pict include exploring the practical applications of π-Calculus type systems and ensuring efficient implementation of π-Calculus primitives such as process creation and communication. This focus on type systems is evident in Pict's support for features like polymorphic channels and higher-order subtyping, which enhance its ability to model complex type interactions within concurrent systems [80].

One of the unique aspects of Pict is its commitment to using the π-Calculus as a practical foundation for programming language design. This involves translating high-level language features directly into π-Calculus constructs, thereby maintaining a close alignment between the language's theoretical basis and its practical implementation. Pict's development also includes experiments with various type system features, such as advanced type relationships or so-called subtyping and polymorphism, to extend the expressive power of the π-Calculus and support advanced programming paradigms within

a concurrent computing context [80].

Below is an excerpt of code written in Pict:

```
cons?*[hd tl r] = (new l (r!l | l?*[n c] = c![hd tl]))

(new r1 (nil![r1] |
         r1?e = (new r2 (cons![33 e r2] |
                         r2?l = ...))))
```

Listing 2.1: Pict code excerpt [80]

This Pict code excerpt previews how to create and use a `cons` cell, which is a basic part of a `list`. The first part sets up a `cons` cell, taking `hd` (head), `tl` (tail), and `r` (result channel). It creates a new channel `l`, sends `l` back through `r`, and listens on `l` to send back the head and tail. The second part creates a `list` by first making an `empty list` using `nil`, sending the result to `r1`. It then gets this `empty list` into `e`, makes a new `cons` cell with `33` as the head and `e` as the tail, and sends the result to `r2`. This example shows how Pict uses `channels` to build and connect `list elements` [80].

2.5 AI

AI represents a broad and rapidly evolving field of technology that aims to simulate human intelligence through machines. It is categorized into several types, each distinguished by its methodologies and applications. Traditional ML forms the foundation, where algorithms learn from and make predictions based on data. These methods are often simpler, relying on statistical techniques to interpret patterns and insights. DL, a subset of ML, utilizes neural networks with many layers to process data in complex ways, enabling it to handle tasks like image and speech recognition more effectively. The newest frontier, Gen AI, goes further by not just analyzing data but also creating new content that mimics human outputs, such as text, images, and audio. Each type of AI has unique strengths, making them suited to different tasks and revolutionizing diverse sectors from healthcare to entertainment [42, 16].

Together, these AI technologies are not only enhancing existing applications but are also creating new opportunities for innovation that were previously unimaginable, setting the stage for future advancements that could further redefine what machines are capable of achieving.

2.5.1 Gen AI and rise of agentic workflows

Gen AI is a transformative technology that leverages advanced ML models to create new content, such as text, images, audio, and video, based on the data it has been trained on. This capability is revolutionizing various business processes by automating complex

and time-consuming tasks, thereby enhancing efficiency and productivity. Gen AI models, such as Generative Adversarial Network (GAN) and Large Language Model (LLM) like GPT-3 [129], are particularly adept at generating high-quality, contextually relevant content, which can be used in applications ranging from customer service to content creation and beyond. Training these sophisticated models, however, requires significant computational resources [15].

The process involves extensive data processing and continuous iterations, necessitating powerful hardware and substantial energy, which can lead to high operational costs and environmental considerations. As the demand for such technology grows, there is a parallel push towards developing more energy-efficient AI systems and optimizing the algorithms to reduce the overall resource footprint. These efforts are critical in ensuring that the benefits of Gen AI can be harnessed sustainably and ethically, supporting its broader adoption and innovation across various sectors [16, 15].

2.6 Cloud computing

Cloud computing offers the on-demand delivery of essential computing services such as servers, storage, databases, networking, software, and analytics through the internet. This innovative model eliminates the need for direct management of physical hardware, enabling users to efficiently access and utilize these resources. Key benefits of cloud computing include cost savings, scalability, flexibility, and efficiency. Users benefit financially by paying only for the services they consume, which also allows them to scale resources flexibly to match fluctuating demands [4].

The distinct advantages of cloud computing are demonstrated by essential features such as on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. These features collectively facilitate the automatic provisioning of resources, ensure service access via any device, enable resource sharing among multiple users, allow for the agile scaling of resources, and support the monitoring and optimization of resource usage [34].

Several major companies dominate the cloud computing market, each offering a range of services tailored to different business needs [53]:

- Amazon Web Services (AWS): AWS is the largest cloud service provider, known for its extensive range of services and global reach. It is particularly strong in public cloud offerings and provides robust solutions for computing, storage, and databases.

- Microsoft Azure: Azure is a leading cloud platform that offers a wide array of services, including developer tools, machine learning, and Internet of Things (IoT) solutions. It is highly regarded for its integration with Microsoft products and services, making it a popular choice for enterprises.

- Google Cloud Platform (GCP): GCP is known for its strong capabilities in data analytics, ML, and AI. It offers a suite of tools for building, deploying, and scaling applications, and is favored for its advanced AI and ML services.

Cloud computing emerges as a pivotal resource in training Gen AI models, offering the necessary computational power and storage solutions that are often too costly or complex for on-site setups. Services from major cloud platforms such as AWS, Azure, and GCP are crucial in this area, providing access to advanced GPUs and TPUs, alongside environments specially tailored for ML tasks. Tools like AWS SageMaker, Google Cloud Vertex AI Platform, and Azure ML stand out by offering robust frameworks for not only building and training but also deploying AI models efficiently [118, 53].

By utilizing cloud computing, organizations gain the ability to scale and accelerate the development and deployment of Gen AI technologies. This scalability is essential for managing large datasets and complex computations. Furthermore, the flexibility of cloud services supports continuous experimentation with various model configurations and adjustments. Enhanced features, including automated hyperparameter tuning, model versioning, and integrated data pipelines, substantially simplify and optimize the AI development life cycle. Thus, cloud computing plays a vital role in meeting the intense demands of advanced Gen AI applications, providing an integrated infrastructure that facilitates innovation and efficiency in AI projects [126, 16].

2.7 Technologies

The following sections explore in more detail the technologies that were critical to the development of the research artifact, examining their contributions and significance. ANother Tool for Language Recognition (ANTLR) is a tool used for the development of language and grammar parsers and was employed in the artifact for programming language specification [128]. The SPADE framework [70], a MAS framework, served as the foundation for initiating agent communication. Additionally, sockets and specifically Netcat [46] are discussed in greater depth, as communication occurs between distributed agents over the internet.

2.7.1 ANTLR

ANTLR is a technology that provides a tool for developing parsers capable of consuming (or reading), processing, executing, and translating structured text and binary files. This technology is particularly useful for parsing various specifications, such as Docker Compose [110] or Helm Chart [128] specifications. The development of a parser with ANTLR involves multiple aspects and steps: grammar specification, parser generation, and business logic implementation [75].

To enable a parser to identify key tokens and phrases, it is necessary to specify a grammar that follows patterns defined by ANTLR. This grammar is used to define various tokens and rules that the parser should focus on during parsing. Tokens represent a single character or a sequence of characters divided by whitespace, while rules can be seen as specifying the order in which tokens appear, effectively acting as phrases. Defining rules, in addition to tokens, is crucial for providing context to the parser's operation, such as understanding whether a token is used within a creation or an update context. Based on the defined grammar, ANTLR generates a parser class comprised of start and end methods for each rule in the grammar. The start method allows an engineer to implement logic when the parser enters a rule, while the end method enables the implementation of business logic upon the completion of a rule. If a rule consists of multiple sub-rules, the parser generates methods for these as well. This results in a tree structure of methods that are traversed during runtime, each method provided with a context containing additional information to assist the engineer in understanding the parser's position when parsing a specification [65, 74].

ANTLR stands out by integrating the description of lexical and syntactic analysis, accepting grammars for languages with extended BNF notation, and automatically generating abstract syntax trees. It generates human-readable recursive-descent parsers in C or C++ from LL(k) grammars, supporting predicates that allow semantic and syntactic context to direct the parse systematically [75]. These predicates eliminate the need for manual adjustments to the parser output, even for challenging parsing problems. ANTLR's parsers are easy to design and debug, even for non-parsing experts, due to the clear correspondence between the grammar specification and the output [74].

The methods in ANTLR can be used in various ways, from implementing business logic executed as the parser iterates through the input data, to translating captured data into formats more suitable for further processing by different systems. A parser generated from ANTLR, extended with an engineer's business logic, can consume any specification provided to it. Below is an example of a grammar snippet comprised of simple tokens and rules [65]:

```
1  grammar Calc;
2
3  // entry point
4  calc:   expr+ EOF;
5
6  // expressions
7  expr:   expr ('+'|'-') expr
8      |   INT
9      ;
10
11 // rules
12 INT :   [0-9]+;
```

```
13 WS  :   [ \t\r\n]+ -> skip;
```

Listing 2.2: ANTLR grammar example

ANTLR will generate the following listener class from the grammar:

```
1  class CalcListener(ParseTreeListener):
2
3      # Enter a parse tree produced by CalcParser#calc.
4      def enterCalc(self, ctx:CalcParser.CalcContext):
5          pass
6
7      # Exit a parse tree produced by CalcParser#calc.
8      def exitCalc(self, ctx:CalcParser.CalcContext):
9          pass
10
11
12     # Enter a parse tree produced by CalcParser#expr.
13     def enterExpr(self, ctx:CalcParser.ExprContext):
14         pass
15
16     # Exit a parse tree produced by CalcParser#expr.
17     def exitExpr(self, ctx:CalcParser.ExprContext):
18         pass
```

Listing 2.3: ANTLR listener class

2.7.2 Network sockets

In CS, a socket is a fundamental concept used in network communications, serving as an endpoint in a two-way communication link between two programs running on a network. Sockets are managed through a software interface provided by the operating system, which allows applications to send and receive data bi-directionally in real-time. This mechanism is crucial for developing network applications where two or more devices need to communicate over a network, such as the internet or a private local area network [39].

Sockets operate at the transport layer and can be created using different protocols, primarily Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP sockets, known as stream sockets, provide a reliable, connection-oriented service, ensuring that data sent is received by the target device in the same order it was sent. In contrast, UDP sockets, or datagram sockets, offer a connectionless service without guaranteeing the order or delivery of packets, which can be faster but less reliable [39, 45].

A socket is identified by an IP address combined with a port number, which together allow the network to direct the incoming and outgoing data to the correct programs. Typically, a server program will listen on a specific port and accept incoming connections from

clients, which specify the server's IP address and port number to establish a connection [47].

Netcat is a versatile command-line utility that reads and writes data across network connections using TCP or UDP. The tool is invaluable for network debugging and exploration, providing a wide range of functions from port scanning and listening to transferring files and creating backdoors [46].

Netcat's simplicity and power come from its ability to easily create any kind of connection that can be needed in a network. Common uses include [22, 32]:

- Port scanning: Checking for open ports on a target machine.
- File transfer: Sending files across a network using a simple command.
- Chat server: Setting up a minimal chat server where messages can be sent and received in real-time.
- Connection testing: Verifying that a server is running and accessible over the network.

Netcat operates in two main modes: the connect mode, where it acts as a client initiating connections to servers, and the listen mode, where it acts as a server waiting for incoming connections. This flexibility makes it an essential tool for system administrators and network engineers for testing and troubleshooting network configurations [32, 46].

2.7.3 SPADE

SPADE is a Python framework designed for the development of MASs. It is based on the Extensible Messaging and Presence Protocol (XMPP), which facilitates communication between agents [93]. This technology enables developers to set up and implement the business logic of individual agents, which may be part of the same or different MASs. Agents in SPADE communicate over XMPP, a messaging protocol known for its features supporting instant messaging, multi-party communication, and message metadata specification. XMPP protocols are public, open, free, secure, and standardized. They also offer decentralization, as anyone can host an XMPP server to facilitate message-passing throughput [88, 93].

SPADE incorporates asynchronous concepts and is based on `asyncio` [40]. In SPADE, an agent is implemented as a class that can encompass various functionalities such as agent startup logic, agent tear-down logic, and agent business logic. The business logic is implemented as a behavior, and SPADE offers multiple behavior types including one-off, periodic, cyclic, and Finite State Machine (FSM). A one-off behavior is designed to execute only once, and the agent may shut down upon its completion if configured to do so. Periodic behavior executes a defined number of times at specified intervals, functioning

like a cron job. Cyclic behavior repeats its business logic in continuous cycles, akin to a while loop, and is often used for communication, particularly when awaiting incoming messages. FSM behavior introduces different states and transitions, allowing for varied logic execution depending on the current state [70, 69].

Agents in SPADE can implement a wide range of complex behaviors, with the flexibility to incorporate various external libraries compatible with the SPADE ecosystem. An example snippet of an agent implemented using SPADE 3.0 [70] is shown below:

```
class MyAgent(Agent):
    class CyclicBehavior(behaviour.CyclicBehaviour):
        async def run(self):
            print("This is a cyclic behaviour.")
            await asyncio.sleep(1)  # Pause for a second

    class OneShotBehavior(behaviour.OneShotBehaviour):
        async def run(self):
            print("This is a one-shot behaviour.")

    class PeriodicBehavior(behaviour.PeriodicBehaviour):
        async def run(self):
            print("This is a periodic behaviour.")
            await asyncio.sleep(2)  # Pause for two seconds

    async def setup(self):
        print("Agent starting...")
        cyclic_behaviour = self.CyclicBehavior()
        one_shot_behaviour = self.OneShotBehavior()
        periodic_behaviour = self.PeriodicBehavior(period=3)  # Executes
            every 3 seconds

        self.add_behaviour(cyclic_behaviour)
        self.add_behaviour(one_shot_behaviour)
        self.add_behaviour(periodic_behaviour)
```

Listing 2.4: SPADE agent comprised with multiple different behaviors

Additionally, it is important to highlight the role of metadata specification in the message-passing mechanism. Each message sent from an agent can include custom metadata, which may be used to specify the message ontology. On the receiving side, an agent can define filters based on metadata values to selectively receive only relevant messages. Furthermore, metadata can be leveraged not only for filtering messages but also for binding different behaviors or business logic based on metadata values [69]:

```
class AgentA(Agent):
    class SendBehavior(OneShotBehaviour):
        async def run(self):
```

```
4             msg = Message(to="agent-b@agent.com")  # Target Agent B
5             msg.set_metadata("performative", "query-ref")
6             msg.set_metadata("ontology", "APiQuery")
7             msg.body = "Requesting information."
8
9             await self.send(msg)
10            print("Agent A has sent a message to Agent B.")
11
12    async def setup(self):
13        print("Agent A started")
14        self.add_behaviour(self.SendBehavior())
15
16class AgentB(Agent):
17    class ReceiveBehavior(CyclicBehaviour):
18        async def run(self):
19            msg = await self.receive(timeout=10)  # Wait for a message
                  for 10 seconds
20                if msg:
21                    print("Agent B received a matching message.")
22                    # Process the message
23
24    async def setup(self):
25        print("Agent B started")
26        rb_template = Template(
27            metadata={"performative": "query-ref", "ontology": "APiQuery
                  "}
28        )
29        self.add_behaviour(self.ReceiveBehavior(), rb_template)
```

Listing 2.5: Communication between SPADE agents with specified metadata

Chapter 3

Frameworks and libraries for MAS architecture

This chapter provides a comprehensive analysis of several widely recognized open-source frameworks and libraries, with the objective of understanding their respective advantages and disadvantages. This detailed examination is intended to inform the focus areas for the development of the artifact. The selection of these frameworks and libraries is based on their current popularity and relevance within the domain of Multi-agent system (MAS). Notably, some of these tools are relatively recent additions, emerging in response to advancements in Generative Artificial Intelligence (Gen AI), which have significantly influenced the evolution of MAS architectures. Consequently, while it is challenging to definitively determine their long-term popularity and utility, this chapter evaluates these frameworks and libraries to assess the following components:

- Support for distributed agents: Evaluates the framework's ability to manage and coordinate agents across networked environments.
- Communication flows specification: Assesses the framework's ability to specify and manage which agents communicate with each other and how they exchange information.
- Integration into existing systems: This component assesses the framework's ease of integration with existing agent systems, focusing on compatibility and adaptability to ensure seamless incorporation with minimal modifications or disruptions.
- Modularity: Evaluates how easily the framework allows modification or replacement of existing agents, facilitating flexible system updates and enhancements.
- Performance: Assesses the system's emphasis on readable code, robust error handling, and efficient coding practices, focusing on minimizing redundancy, reducing potential failure points, and optimizing resource utilization.

The following libraries and frameworks are analyzed:

- Smart Python Agent Development Environment (SPADE) [108]
- Microsoft AutoGen [6]
- LangGraph [50]
- Crew AI [82]
- EveJS [23]
- JADE [49]
- Docker [110]
- Kubernetes [18]

3.1 SPADE

SPADE is a robust framework designed for developing distributed agents, leveraging the Extensible Messaging and Presence Protocol (XMPP) protocol to facilitate seamless communication across networked environments. It requires a specific architectural setup where agents are defined as classes with distinct behaviors, promoting modularity and clarity in agent design. While SPADE's architecture necessitates careful setup and adherence to its communication protocols, it offers a well-structured approach to MAS development. This framework supports isolated agent execution, enhancing system resilience by ensuring that failures in individual agents do not impact others. Despite these challenges, SPADE provides a comprehensive solution for building scalable and resilient MASs [70, 69, 93].

The following code, written with SPADE, demonstrates two agents communicating over the network. The agents are shown together in the same file for preview purposes, but can be separated into different files and deployed on different servers:

```
# agent_a.py hosted on a server in Europe
class AgentA(Agent):
    class SendBehavior(PeriodicBehaviour):
        async def run(self):
            msg = Message(to="agent-b@agent.com")
            current_timestamp = datetime.now().isoformat()
            msg.body = f"Currently it is: {current_timestamp}"

            await self.send(msg)

    async def setup(self):
        self.add_behaviour(self.PeriodicBehavior(period=60))
```

```
13
14 # agent_b.py hosted on a server in the US
15 class AgentB(Agent):
16     class ReceiveBehavior(CyclicBehaviour):
17         async def run(self):
18             msg = await self.receive(timeout=5)
19             if msg:
20                 print(f"Current time is {msg}")
21
22     async def setup(self):
23         self.add_behaviour(self.ReceiveBehavior())
```

Listing 3.1: Communication between distributed SPADE agents

Below are observations on the framework components being analyzed [70, 69, 93]:

- Support for distributed agents: SPADE operates on top of the XMPP network protocol, which inherently supports the management and communication of distributed agents. This integration with XMPP facilitates robust and scalable multi-agent interactions across diverse network environments. However, a notable limitation is that agents developed with SPADE are restricted to using XMPP exclusively, limiting flexibility in adopting alternative communication protocols. This constraint necessitates careful consideration when integrating SPADE into systems that may require different or multiple communication standards.

- Communication flows specification: In SPADE, each agent can specify listeners that define the expected source and format (ontology) of incoming messages, ensuring precise communication. Agents can also specify target agents for outgoing messages, ensuring directed communication flows. However, SPADE does not support broadcasting to multiple agents simultaneously, limiting communication to specified individual agents only.

- Integration into existing systems: SPADE introduces a structured approach to developing agents as classes, each implementing one or more behaviors, promoting modularity and clarity. Communication between agents requires a network connection and proper setup, adding configuration steps. While SPADE offers a robust architecture and clear conventions for MAS development, migrating existing code and business logic to SPADE may be time-consuming. This transition involves adapting systems to fit SPADE's communication protocols and behavioral architecture, potentially requiring significant code refactoring and integration efforts.

- Modularity: SPADE's architecture facilitates smooth replacement of existing business logic within an agent, as long as the expected flow of logic—such as whether it

functions as a listener or a one-off behavior—remains intact. This design allows for easy updates and modifications to an agent's internal logic without disrupting overall system functionality. Additionally, updating the communication flow between agents is straightforward, requiring only adjustments to the communication contract, which includes the agent's address and possibly the ontology description. This modular approach simplifies maintenance and scalability, allowing developers to efficiently manage and evolve MASs.

- Performance: As previously mentioned, SPADE requires significant architectural and code changes, which must be replicated for all agents, leading to potential code redundancy and increased risk of failure points. However, the isolation of agents, each running as a separate thread, ensures that failures in one agent do not impact the performance of others. This isolation enhances system resilience, allowing individual agent failures to be contained without affecting the overall system stability. Despite the initial setup overhead, this threading model contributes to robust performance management within MASs.

3.2 Microsoft AutoGen

Microsoft Autogen provides an advanced multi-agent conversation framework designed for next-generation Large Language Model (LLM) applications, focusing on collaboration, teachability, and personalization. This open-source framework enables users to build sophisticated LLM workflows with modular agents and conversation-based programming, simplifying development and enhancing code reuse. Autogen's approach allows multiple agents to learn and collaborate independently, significantly reducing user effort. Key benefits include support for diverse LLM configurations, native tool usage through code generation and execution, and the inclusion of a Human Proxy Agent, which facilitates seamless integration of human feedback and involvement at various stages [124].

The following code, written in AutoGen, encompasses multiple agents powered by LLM models:

```
config_list = autogen.config_list_from_json(
    "OAI_CONFIG_LIST",
    filter_dict={
        "model": ["gpt-4", "gpt-4-0314", "gpt4", "gpt-4-32k", "gpt-4-32k-0314", "gpt-4-32k-v0314"],
    },
)

llm_config = {"config_list": config_list, "cache_seed": 42}
user_proxy = autogen.UserProxyAgent(
    name="User_proxy",
```

```
    system_message="A human admin.",
    code_execution_config={
        "last_n_messages": 2,
        "work_dir": "groupchat",
        "use_docker": False,
    },  # Please set use_docker=True if docker is available to run the generated code. Using docker is safer than running the generated code directly.
    human_input_mode="TERMINATE",
)
coder = autogen.AssistantAgent(
    name="Coder",
    llm_config=llm_config,
)
pm = autogen.AssistantAgent(
    name="Product_manager",
    system_message="Creative in software product ideas.",
    llm_config=llm_config,
)
groupchat = autogen.GroupChat(agents=[user_proxy, coder, pm], messages=[], max_round=12)
manager = autogen.GroupChatManager(groupchat=groupchat, llm_config=llm_config)

user_proxy.initiate_chat(
    manager, message="Find a latest paper about gpt-4 on arxiv and find its potential applications in software."
)
```

Listing 3.2: Multi-agent communication in Microsoft AutoGen

Below are observations on the framework components being analyzed [124, 6]:

- Support for distributed agents: The AutoGen framework is primarily designed for use within a single system, meaning it does not support distributing agents across multiple environments. It features a graph-like approach for defining inter-agent communication, allowing users to visually map out and manage interactions between agents. This centralized design ensures cohesive operation and simplifies the management of agent workflows, but it limits the framework's ability to scale across diverse, distributed systems. By utilizing this graph-like structure, developers can easily specify and adjust communication pathways, enhancing the clarity and efficiency of agent interactions within the system.

- Communication flows specification: This framework enables the specification of agent pairs involved in communication through a programmatically defined, graph-

like structure, rather than relying on explicit graph definitions. This dynamic approach allows agents to determine their communication targets in real-time, providing flexibility in managing interactions. However, this method can become cumbersome in environments with a large number of agents, particularly if the logic for determining communication paths is complex. As the number of agents increases, managing and optimizing these communication flows can become increasingly challenging, potentially leading to inefficiencies and difficulties in maintaining clear and effective interaction protocols. This limitation highlights the need for careful planning and robust logic design when implementing AutoGen in more extensive multi-agent systems.

- Integration into existing systems: AutoGen seamlessly integrates with systems that utilize LLMs, simplifying the migration process by focusing primarily on defining communication flows. This integration allows for the efficient incorporation of LLMs into existing systems, streamlining the adaptation of current code to leverage AutoGen's capabilities. However, the framework is optimized for LLM, which may pose challenges when integrating other types of Gen AI models or diverse business logic that adhere to different communication protocols or contracts. This could potentially lead to complicating the integration process and limiting the framework's flexibility in accommodating a broader range of Artificial Intelligence (AI) models and business logic.

- Modularity: This framework offers flexibility by allowing the agents involved in communication, along with their corresponding business logic (prompts), to be easily updated or replaced. This adaptability facilitates rapid adjustments to agent behaviors and communication strategies, enabling developers to quickly respond to changing requirements or optimize performance. The ease of updating or replacing agents and their logic supports continuous improvement and experimentation, enhancing the overall robustness and efficiency of the MAS.

- Performance: Introducing Autogen into existing LLM-powered applications requires minimal changes, primarily involving the restructuring of agent logic and the specification of communication flows. The framework's design encapsulates repetitive logic and error handling, which can lead to a reduction in code size for the migrated applications. This encapsulation not only streamlines the codebase but also enhances control, monitoring, and failure management, making the system more robust and easier to maintain. By simplifying these aspects, Autogen facilitates a smoother transition and improves the overall efficiency and reliability of the MAS, allowing developers to focus more on enhancing functionalities rather than managing complex code structures.

3.3 LangGraph

LangGraph is a powerful extension of the LangChain products family, designed to enhance the creation and management of agent runtimes by introducing cyclical graphs. Unlike traditional directed acyclic graphs, LangGraph supports cyclic behaviors, which are essential for complex applications involving LLM. LangGraph is particularly effective for complex tasks that can be broken down into smaller, manageable parts handled by different agents. This framework utilizes a graph-based structure to define agents and their interactions, where each agent is represented as a node, and edges define the communication paths between them. Each agent in a graph has their own state but also has the capability to update the central state object shared between agents [50]. The following code snippet, written using LangGraph, demonstrates how a graph describing relationships is specified:

```
def agent_node(state, agent, name):
    result = agent.invoke(state)
    # We convert the agent output into a format that is suitable to
        append to the global state
    if isinstance(result, ToolMessage):
        pass
    else:
        result = AIMessage(**result.dict(exclude={"type", "name"}), name
            =name)
    return {
        "messages": [result],
        # Since we have a strict workflow, we can
        # track the sender so we know who to pass to next.
        "sender": name,
    }

llm = ChatOpenAI(model="gpt-4-1106-preview")

# Research agent and node
research_agent = create_agent(
    llm,
    [tavily_tool],
    system_message="You should provide accurate data for the
        chart_generator to use.",
)
research_node = functools.partial(agent_node, agent=research_agent, name
    ="Researcher")

# chart_generator
chart_agent = create_agent(
    llm,
    [python_repl],
```

```
    system_message="Any charts you display will be visible by the user.",
)
chart_node = functools.partial(agent_node, agent=chart_agent, name="chart_generator")

workflow.add_node("Researcher", research_node)
workflow.add_node("chart_generator", chart_node)
workflow.add_node("call_tool", tool_node)

workflow.add_conditional_edges(
    "Researcher",
    router,
    {"continue": "chart_generator", "call_tool": "call_tool", "__end__": END},
)
workflow.add_conditional_edges(
    "chart_generator",
    router,
    {"continue": "Researcher", "call_tool": "call_tool", "__end__": END},
)

workflow.add_conditional_edges(
    "call_tool",
    # Each agent node updates the 'sender' field
    # the tool calling node does not, meaning
    # this edge will route back to the original agent
    # who invoked the tool
    lambda x: x["sender"],
    {
        "Researcher": "Researcher",
        "chart_generator": "chart_generator",
    },
)
workflow.set_entry_point("Researcher")
graph = workflow.compile()
```

Listing 3.3: Specifying agents' communication in LangGraph

The following are observations on the analyzed components of LangGraph [50]:

- Support for distributed agents: LangGraph is capable of supporting multiple agents within a single system environment, where each agent operates independently with its own memory and business logic. These agents are represented as nodes within a directed graph, facilitating potential for distributed operations. However, while

LangGraph inherently supports the orchestration of these nodes within a single system, it lacks built-in capabilities for inter-system communication, limiting agent interaction across different environments. This design choice emphasizes system internal cohesion over external integrations.

- Communication flows specification: The framework utilizes a directed cyclic graph to define dynamic workflows, enabling the specification of complex agent interactions and decision-making processes. The graph's structure is consolidated within a single section of the codebase, enhancing maintainability and scalability. This centralized definition allows for real-time adjustments during runtime, adapting the workflow to evolving conditions and requirements.

- Integration into existing systems: LangGraph is particularly well-suited for integration with systems that already leverage Gen AI technologies, given its design to seamlessly connect with such platforms. This makes it an ideal choice for enhancing systems with advanced AI capabilities without significant overhauls. However, integration with traditional AI or legacy systems might require additional bridging logic, potentially complicating integrations.

- Modularity: The modularity of LangGraph is a standout feature, facilitating easy modifications and extensions within the application. Users can replace agents, alter interaction pathways, and adjust dependencies through a single graph definition file. This not only simplifies updates and maintenance but also encourages experimentation and rapid development cycles. The architecture's flexibility supports evolving business needs and technological advancements without extensive redevelopment.

- Performance: LangGraph optimizes performance by minimizing the boilerplate code necessary for defining business logic and graph structures. By abstracting less critical coding aspects and focusing on the graph and its logic, the framework enhances clarity and efficiency. Additionally, LangGraph supports supplementary features like monitoring and diagnostics tools, which are crucial for maintaining performance at scale. These features ensure that LangGraph can handle increasing workloads and complexity, making it a robust solution for growing applications.

3.4 CrewAI

CrewAI is a multi-agent framework that facilitates the orchestration and management of groups of autonomous AI agents. Operating within the LangChain ecosystem [50], it utilizes a role-based design that assigns specific roles, goals, and tools to each agent, allowing for highly efficient task performance. This framework enables autonomous inter-agent delegation, where agents can dynamically allocate tasks among themselves to enhance

problem-solving capabilities and operational efficiency. CrewAI features a modular architecture composed of key elements such as Agents, Tasks, Tools, and Crews, which synergize to form a cohesive and high-performing team. The framework accommodates both sequential and hierarchical task processes and is continuously being developed to incorporate more sophisticated coordination strategies [82].

Following is a snippet of CrewAI code that defines previews how agents and tasks are implemented:

```
search_tool = SerperDevTool()

# Define your agents with roles and goals
researcher = Agent(
  role='Senior Research Analyst',
  goal='Uncover cutting-edge developments in AI and data science',
  backstory="""You work at a leading tech think tank.
  Your expertise lies in identifying emerging trends.
  You have a knack for dissecting complex data and presenting actionable
      insights.""",
  verbose=True,
  allow_delegation=False,
  tools=[search_tool]
  # You can pass an optional llm attribute specifying what model you
     wanna use.
  # It can be a local model through Ollama / LM Studio or a remote
  # model like OpenAI, Mistral, Antrophic or others (https://docs.crewai
     .com/how-to/LLM-Connections/)
  #
  # import os
  # os.environ['OPENAI_MODEL_NAME'] = 'gpt-3.5-turbo'
  #
  # OR
  #
  # from langchain_openai import ChatOpenAI
  # llm=ChatOpenAI(model_name="gpt-3.5", temperature=0.7)
)
writer = Agent(
  role='Tech Content Strategist',
  goal='Craft compelling content on tech advancements',
  backstory="""You are a renowned Content Strategist, known for your
     insightful and engaging articles.
  You transform complex concepts into compelling narratives.""",
  verbose=True,
  allow_delegation=True
)

# Create tasks for your agents
```

```
task1 = Task(
  description="""Conduct a comprehensive analysis of the latest
      advancements in AI in 2024.
  Identify key trends, breakthrough technologies, and potential industry
       impacts.""",
  expected_output="Full analysis report in bullet points",
  agent=researcher
)

task2 = Task(
  description="""Using the insights provided, develop an engaging blog
  post that highlights the most significant AI advancements.
  Your post should be informative yet accessible, catering to a tech-
      savvy audience.
  Make it sound cool, avoid complex words so it doesn't sound like AI.
      """,
  expected_output="Full blog post of at least 4 paragraphs",
  agent=writer
)

# Instantiate your crew with a sequential process
crew = Crew(
  agents=[researcher, writer],
  tasks=[task1, task2],
  verbose=2, # You can set it to 1 or 2 to different logging levels
)

# Get your crew to work!
result = crew.kickoff()
```

Listing 3.4: Specifying agents and tasks in CrewAI

Below are descriptions of how the analyzed components function within the CrewAI ecosystem [82]:

- Support for distributed agents: CrewAI operates within a singular process and does not currently support distributed agents. All agents must reside and operate within the same system environment, which may limit scalability across distributed systems.

- Communication flows specification: CrewAI places significant emphasis on defining agents, roles, and tasks. Each agent, depending on its assigned role, might execute the same tasks but produce different outputs based on that role. The framework's architecture incorporates hierarchical elements reminiscent of holonic systems [31]. However, unlike some other frameworks that define communications at a holistic level, CrewAI specifies communication at more granular levels, which can affect

how information flows between agents.

- Integration into existing systems: Framework can be adapted to a wide variety of applications due to its extensive support for MAS concepts. However, integration often requires significant modifications or restructuring of existing business logic, which may not be straightforward and could necessitate considerable development effort.
- Modularity: CrewAI's design is highly modular, accommodating not just generic AI applications but also specific business logic needs. It allows for flexible technology usage and makes it relatively smooth to replace or extend agents and their tasks. However, these modifications are not centralized and must be implemented wherever the agents or tasks are specifically defined within the codebase.
- Performance: Architecture of this framework supports extensive features that allow for control at multiple levels, potentially increasing the complexity of the system. This complexity might lead to more extensive coding requirements and a greater possibility of failures. Notably, CrewAI's design does not specifically incorporate multithreading, which means that failures in one agent could be more impactful, potentially affecting the system's overall robustness and responsiveness.

3.5 EveJS

EveJS is a sophisticated framework designed for the development and management of distributed systems using software agents. This framework supports agent-based modeling, allowing agents to autonomously perform tasks, communicate, and make decisions. It is inherently cross-platform, functioning seamlessly across environments like Node.js and web browsers, which ensures flexibility in deployment. EveJS emphasizes communication flexibility, facilitating agent interaction across various protocols such as Hypertext Transfer Protocol (HTTP), AMQP, and WebSocket (WS). Additionally, its modularity allows for the extension of agent capabilities with different communication patterns like request-reply and publish-subscribe. This setup enables agents within EveJS to discover and communicate with each other, either directly or through mediated connections, making it an effective tool for creating distributed applications that require robust, decentralized processes [23, 68].

The following snippet of EveJS code demonstrates how agents and tasks are implemented:

```
1 var eve = require('evejs');
2 var WebSocketTransport = require('evejs/dist/eve.custom');
3
4 function MyWebSocketAgent(id) {
```

```
5     // Execute super constructor
6     eve.Agent.call(this, id);
7
8     // Extend the agent with WebSocket transport capabilities
9     this.extend(new WebSocketTransport());
10
11    // Connect to a WebSocket server
12    this.connect('ws://localhost:8080/agents/' + this.id);
13  }
14
15  // Extend the Agent prototype
16  MyWebSocketAgent.prototype = Object.create(eve.Agent.prototype);
17  MyWebSocketAgent.prototype.constructor = MyWebSocketAgent;
18
19  // Override the receive method to handle incoming messages
20  MyWebSocketAgent.prototype.receive = function (from, message) {
21    console.log(from + ' said: ' + message);
22  };
23
24  // Create an instance of the agent
25  var agent = new MyWebSocketAgent('myAgent1');
26
27  // Example of sending a message
28  agent.send('ws://localhost:8080/agents/agent2', 'Hello Agent 2!');
```

Listing 3.5: Agents implementation in EveJS

Below are descriptions of the analyzed components of EveJS [23]:

- Support for distributed agents: EveJS facilitates the management of distributed agents over multiple protocols including AMQP, WS, and HTTP. This framework's interoperability ensures that agents can operate independently of the framework itself, providing versatility in how they are deployed and interact within different system architectures.

- Communication flows specification: Each agent in EveJS is responsible for configuring its own listeners and communication mechanisms. While the framework allows for considerable flexibility in defining these flows, it does not natively support advanced metadata specification within messages. This design choice prioritizes flexibility and customization over standardized communication formats.

- Integration into existing systems: Implementing EveJS within an existing system necessitates a thoughtful approach, particularly with the setup of agent interfaces. Although integration requires careful planning and execution, the business logic of the existing system can generally be preserved and encapsulated within new or

existing agents. This modular encapsulation helps maintain system integrity while integrating new functionalities.

- Modularity: The framework's modularity and flexibility are pivotal when updating communication flows or business logic. EveJS's design supports easy updates to communication protocols and logic, making it highly adaptable to evolving technological needs or project requirements. This is particularly valuable in dynamic environments where system requirements can change rapidly.

- Performance: Each agent in EveJS operates independently, which isolates them from failures in other parts of the system. This isolation enhances system robustness by ensuring that issues in one agent do not propagate to others. However, this can lead to some redundancy in code, particularly in the specification of communication mechanisms. While this redundancy may increase the resource footprint, it also allows for tailored configurations that optimize agent interactions based on specific operational needs.

3.6 JADE

Java Agent DEvelopment Framework (JADE) is a robust Java-based framework designed to facilitate the development of MASs in compliance with Foundation for Intelligent Physical Agents (FIPA) standards. FIPA compliance ensures that systems developed with JADE can communicate and interact with other FIPA-compliant agent systems globally, fostering interoperability and standardization in agent communications. JADE provides an extensive set of tools that support the creation, management, and deployment of agents, featuring capabilities such as asynchronous message-passing using Agent Communication Language (ACL) for effective agent interaction. Engineered for high scalability and robustness, the framework is capable of handling thousands of agents across networks or the Internet, enhancing system reliability and performance under various conditions. Additionally, JADE supports agent mobility, allowing agents to move across machines to optimize resource use and balance loads, further contributing to the framework's flexibility and adaptability in dynamic environments. With a comprehensive suite of graphical tools that aid in monitoring and debugging agent behaviors, JADE offers a complete solution for building distributed applications that require collaborative and intelligent agent operations [11, 49].

Code snippet below showcases the usage of JADE to achieve communication between two agents:

```
1 # ReceiverAgent.java
2 import jade.core.Agent;
3 import jade.core.behaviours.CyclicBehaviour;
```

```
4  import jade.lang.acl.ACLMessage;
5
6  public class ReceiverAgent extends Agent {
7      protected void setup() {
8          addBehaviour(new CyclicBehaviour(this) {
9              public void action() {
10                 ACLMessage msg = receive();
11                 if (msg != null) {
12                     System.out.println("Received message: " + msg.
                           getContent());
13                 }
14                 block();
15             }
16         });
17     }
18 }
19
20 # SenderAgent.java
21 import jade.core.Agent;
22 import jade.core.behaviours.OneShotBehaviour;
23 import jade.lang.acl.ACLMessage;
24 import jade.core.AID;
25
26 public class SenderAgent extends Agent {
27     protected void setup() {
28         addBehaviour(new OneShotBehaviour(this) {
29             public void action() {
30                 ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
31                 msg.addReceiver(new AID("receiver", AID.ISLOCALNAME));
32                 msg.setLanguage("English");
33                 msg.setContent("Hello, Receiver!");
34                 send(msg);
35             }
36         });
37     }
38 }
```

Listing 3.6: Agents communication in JADE

The following are descriptions of the analyzed characteristics of JADE [11, 49]:

- Support for distributed agents: JADE is great at handling agents that work across different computers and networks. It allows these agents to work together smoothly on various platforms, making it easier to build and scale up applications. The framework can manage thousands of agents at once over the internet, making it suitable for large and complex systems.
- Communication flows specification: JADE helps agents talk to each other efficiently

using a system called ACL. This language lets agents send messages back and forth, making it possible for them to have detailed interactions, share information, and work together. This is important for agents that need to coordinate and make decisions together.

- Integration into existing systems: The framework can easily fit into existing systems because it works well with standard Java setups. This means you can add JADE to your current projects without major changes. It also follows international standards for agent systems, which helps JADE-based agents communicate with other compatible systems, making it easier to connect different systems together.

- Modularity: The architecture of JADE is designed with modularity in mind, allowing developers to easily adapt and extend the framework to meet specific needs. By breaking down the system into interchangeable components, it offers the flexibility to enhance or modify parts without affecting the overall system. This approach keeps the system well-organized and simplifies updates, making it easier to scale and adapt as project requirements evolve.

- Performance: The framework handles numerous agents simultaneously with minimal impact on system resources. It ensures smooth operation and high responsiveness, even in demanding situations. The mobility feature of agents, which allows them to transfer across machines, optimizes resource usage and balances loads effectively, boosting overall system performance and reliability.

3.7 Docker

Docker uses containerization technology to encapsulate applications and their dependencies into containers, ensuring consistency across various computing environments from development to production. These containers are lightweight and contain everything necessary to run the software, including the application code, runtime environment, libraries, and system tools. Docker Compose, a tool for defining and managing multi-container Docker applications, allows you to use a Yet Another Markup Language (YAML) file to configure your application's services, including their dependencies, networks, and volumes. Additionally, Docker Swarm provides native clustering functionality to manage a cluster of Docker engines, enhancing fault tolerance and scalability. This setup enables you to manage complex applications with ease by defining all components in a single configuration file. Compose ensures that services are launched in an orderly fashion based on their dependencies, maintaining a clear, controlled startup and operational environment. However, while Compose can dictate service availability and startup order, it does not manage the internal communication flows between services; this aspect must be handled

at the application level by developers [110, 56].

```
services:
  database:
    image: mysql:8.0
    container_name: mysql-db
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: mydb
      MYSQL_USER: user
      MYSQL_PASSWORD: password
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - app-network

  app:
    image: python:3.9-slim
    container_name: flask-app
    working_dir: /app
    volumes:
      - ./app:/app
    command: python app.py
    environment:
      FLASK_ENV: development
      DATABASE_HOST: database
      DATABASE_USER: user
      DATABASE_PASSWORD: password
      DATABASE_NAME: mydb
    depends_on:
      - database
    networks:
      - app-network

volumes:
  db_data:

networks:
  app-network:
```

Listing 3.7: Docker Compose YAML configuration

The following is an analysis of the characteristics of Docker [110]

- Support for distributed agents: Docker supports distributed agents in the form of services managed by Docker Compose for single-machine setups and Docker Swarm for multi-node clusters. Docker Swarm enables clustering across multiple physical machines or virtual machines, offering high availability, load balancing, and scaling

capabilities, making it suitable for distributed systems.

- Communication flows specification: Docker operates at an infrastructure level, allowing developers to specify dependencies between services, such as startup order, through Docker Compose. However, it does not handle or dictate the communication flows between services; these must be defined and implemented within the application logic.

- Integration into existing systems: This service orchestration provider can be easily integrated into existing systems, provided the application code is containerized. This process involves creating `Dockerfile` configurations and ensuring dependencies are encapsulated. Docker's broad compatibility with various environments simplifies adoption and enables consistent application behavior across different stages of development and deployment.

- Modularity: Docker's architecture promotes modularity by allowing services in Docker Compose to be easily swapped or replaced. This is achieved as long as the replacement services maintain the same communication contracts, such as API endpoints, ports, and protocols. Since Docker Compose operates at the infrastructure level, it enables seamless updates and iterations without disrupting the broader application ecosystem.

- Performance: Performance optimization is supported in Docker Compose by allowing developers to configure containerized applications with infrastructure-specific adjustments, such as networking and caching. However, Docker's resource allocation features, particularly in Compose and Swarm, are less granular than those of Kubernetes. Kubernetes provides more flexibility in specifying resource limits for containers, making it more suitable for complex or resource-intensive environments [56].

3.8 Kubernetes

Kubernetes, or K8s, is an open-source platform that automates the deployment, scaling, and management of containerized applications. Built to complement container technologies like Docker, Kubernetes orchestrates the deployment of containers across clusters of machines, enabling seamless scaling, high availability, and efficient resource utilization. It integrates smoothly into existing systems, supporting diverse environments such as public, private, and hybrid clouds, while allowing developers to define workloads as pods, deployments, and services. Kubernetes handles distributed services by automating resource scheduling, load balancing, and self-healing, ensuring system stability and

resilience. While it simplifies infrastructure orchestration and scales workloads dynamically, Kubernetes does not dictate communication flows between services, leaving that responsibility to the application logic. Its robust integration with Docker and other container runtimes, coupled with its ability to optimize performance and manage complex workloads, makes Kubernetes a cornerstone for scalable, reliable, and modern application architectures [19, 48].

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: app-namespace

---
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
  namespace: app-namespace
spec:
  ports:
    - port: 3306
  selector:
    app: mysql

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: app-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:8.0
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "example"
            - name: MYSQL_DATABASE
```

```
41                value: "mydb"
42              - name: MYSQL_USER
43                value: "user"
44              - name: MYSQL_PASSWORD
45                value: "password"
46            ports:
47              - containerPort: 3306
48
49  ---
50  apiVersion: v1
51  kind: Service
52  metadata:
53    name: flask-service
54    namespace: app-namespace
55  spec:
56    ports:
57      - port: 80
58        targetPort: 5000
59    selector:
60      app: flask
61    type: LoadBalancer
62
63  ---
64  apiVersion: apps/v1
65  kind: Deployment
66  metadata:
67    name: flask
68    namespace: app-namespace
69  spec:
70    replicas: 1
71    selector:
72      matchLabels:
73        app: flask
74    template:
75      metadata:
76        labels:
77          app: flask
78      spec:
79        containers:
80          - name: flask
81            image: python:3.9-slim
82            env:
83              - name: FLASK_ENV
84                value: "development"
85              - name: DATABASE_HOST
86                value: "mysql-service"
87              - name: DATABASE_USER
```

```
88                value: "user"
89              - name: DATABASE_PASSWORD
90                value: "password"
91              - name: DATABASE_NAME
92                value: "mydb"
93            ports:
94              - containerPort: 5000
```

Listing 3.8: Kubernetes services configration

Characteristics of Kubernetes are described below [19, 48, 56]:

- Support for distributed agents: Kubernetes provides robust support for distributed services by orchestrating containerized workloads across multiple nodes in a cluster. It ensures high availability and fault tolerance through features like automatic load balancing, self-healing, and resource scheduling. Kubernetes is designed to manage distributed systems at scale, making it ideal for multi-agent applications spanning diverse environments.

- Communication flows specification: Service-to-service communication is facilitated through constructs like Services and Ingress, which manage networking and expose workloads internally or externally. While access rules and network policies can be defined, the platform does not manage application-level communication flows. The responsibility for defining and handling inter-service protocols and data exchange lies entirely with the application logic.

- Integration into existing systems: Seamless integration into existing systems and DevOps pipelines is a key strength, with support for container runtimes like Docker and CRI-O, along with compatibility across public, private, and hybrid cloud platforms. However, migrating to the platform often requires significant re-architecting of applications to align with concepts such as pods, deployments, and services, which can involve a steep learning curve.

- Modularity: Kubernetes excels in modularity by abstracting application components into reusable and independent units such as pods and services. Workloads can be updated, replaced, or scaled without impacting other parts of the system, provided they adhere to the defined APIs or communication contracts. This modular design simplifies maintenance, enabling iterative development and rapid deployment.

- Performance: Granular control over resource allocation allows developers to define CPU, memory, and storage limits for individual containers. This flexibility ensures optimal resource utilization and supports scaling based on workload demand. Additionally, self-healing and load-balancing features enhance performance and re-

silience, making the platform well-suited for high-demand, resource-intensive, and large-scale applications.

3.9 Comparison of the analyzed tools

The comparison of different frameworks and libraries developed specifically for agents, or those offering a higher level of abstraction that also supports agent-based systems, highlights the diverse features each provides. These components have been analyzed across several critical dimensions: support for distributed agents, communication flow specification, integration into existing systems, modularity, and performance. The evaluation reveals that while no single tool comprehensively meets all the assessed criteria, the analysis offers a valuable overview of the current landscape. It serves as a strong foundation for understanding industry offerings and best practices.

Tooling focused on the application layer generally performs better in areas such as integration into existing systems, modularity, and communication flow specification. These tools are often designed with flexibility and extensibility in mind, enabling smoother integration into diverse environments and workflows. In contrast, tools operating at the intersection of the infrastructure and application layers emphasize service orchestration and advanced distributed system capabilities. While these tools might be less modular or integrative with existing systems, they bring unique strengths in enabling and managing distributed agent ecosystems.

The findings are summarized in a comparative analysis that uses symbols to indicate the degree of compliance with each evaluated component. The plus symbol (`+`) indicates that a component is fully met, the forward slash (`/`) symbol indicates it is partially met, and the minus symbol (`-`) signifies that the component is not met. This structured approach provides clearer visibility into the strengths of each individual tool and their comparisons against one another.

	Support for Distributed agents	Communication Flows Specification	Integration into Existing Systems	Modularity	Performance
SPADE	+	+	/	+	/
Microsoft AutoGen	-	+	+	+	+
LangGraph	-	+	+	+	+
Crew AI	-	+	+	+	/
EveJS	+	+	/	+	+
JADE	+	+	/	+	+
Docker	+	-	/	/	/
Kubernetes	+	-	/	/	+

Figure 3.1: Evaluation of components across different tools

Based on the analysis of the components of the selected tools commonly used in the industry, it can be concluded that none of the available tools fully meet the expectations and requirements across all five components. This highlights an opportunity for a new contribution that addresses all five components and serves as an argument for the development of a new programming language and its associated declarative engine as part of this research.

Chapter 4

Solution objectives

The development of the artifact of this research is based on a comprehensive analysis of three key areas: scientific research, industry practices, and the requirements of the Orchestration of Hybrid Artificial Intelligence Methods for Computer Games (O HAI 4 Games) project. Each of these areas has contributed unique insights and perspectives, shaping the design and functionality of the artifact.

Firstly, the review of scientific research provided a strong foundation for understanding current advancements and gaps in the field. Most existing studies have primarily focused on the application and protocol layers, with relatively little attention given to the orchestration layer. This identified gap highlights an opportunity for innovation in artifact development, specifically in addressing challenges related to orchestration and integration across layers [24, 33, 3, 17].

Secondly, industry practices were examined through the analysis of various tools and solutions currently used in real-world applications. These tools predominantly focus on the application and infrastructure layers, offering valuable insights into best practices and practical implementations [110, 18, 70]. However, it is evident that there is a lack of support for communications flows specification, especially with regard to distributed agents.

Lastly, the O HAI 4 Games project has played a critical role in shaping the requirements for artifact development. The project provides concrete use cases that were tested for compatibility with the artifact. These use cases not only served as practical benchmarks for evaluating the effectiveness of the artifact but also offered guidance on what features and functionalities are most valuable in real-world scenarios. This alignment ensures that the artifact is both practical and relevant to the specific needs of the project [64].

By combining theoretical insights, practical industry knowledge, and real-world benchmarks, a comprehensive set of requirements was established to guide the design of the artifact. This holistic approach ensures that the resulting solution is both innovative and aligned with real-world needs, enhancing usability and addressing specific use case

demands.

4.1 O HAI 4 Games

O HAI 4 Games is a project that ran from 2020 to its successful completion in 2024. Led by full professor Dr. Markus Schatten and implemented at the Faculty of Organization and Informatics, University of Zagreb, the project aimed to address critical gaps in the orchestration of hybrid Artificial Intelligence (AI) methods for gaming [100].

The central premise of O HAI 4 Games revolved around hybrid AI, an approach that integrates heterogeneous AI methods, both statistical and symbolic, into cohesive and functional systems. Although AI methods have been widely applied in domains such as healthcare, energy, and predictive analytics, their systematic orchestration in gaming has been largely overlooked. The project addressed this gap by developing methodologies and platforms for real-time, scalable, and believable AI systems tailored to the unique challenges of games.

Computer games have long served as effective testing grounds for advancing AI research, from Claude Shannon's early chess algorithms [104] to the groundbreaking AlphaGo [117]. Building on this tradition, O HAI 4 Games focused on orchestration challenges, particularly the integration of diverse AI methods such as Machine Learning (ML), automated planning, and swarm intelligence. Its approach emphasized ensuring that AI behaviors in games remained consistent, coordinated, and immersive.

A significant aspect of the project was the development of four diverse testbeds, each representing a unique use case:

1. A Massively Multiplayer Online Role-playing Games (MMORPGs): This testbed focused on automating game testing in a social environment. Using the high-level interface developed in the ModelMMORPG project [102], AI ensembles were created to test elements such as Non-player character (NPC) behaviors, quests, and combat scenarios.

2. Virtual NPC assistant & gamification: The second testbed aimed to enhance the IMapBook platform [109], a gamified learning environment that combines e-books with interactive games and social interaction. AI ensembles were used to create virtual assistants and discussion moderators, incorporating Natural Language Processing (NLP) for topic extraction and conversation management.

3. Autonomous Vehicles (AVs) – serious gaming: The third testbed explored serious gaming in the context of AVs. By developing AI ensembles for ambient intelligence, the platform simulated complex AV interactions in realistic gaming environments, utilizing either an existing racing game like TORCS [125] or a newly developed 3D game environment.

4. HoloGame platform: The final testbed centered on developing a holographic or volumetric gaming platform named HoloGame [5]. This innovative use case included designing a novel game for the platform, integrating AI ensembles for elements such as NPCs, content generation, and player modeling.

4.2 Requirements

Based on the analysis, the artifact must meet the following requirements to ensure its efficacy, adaptability, and scalability in practical applications:

- AI support: The artifact should support both traditional and modern hybrid methods of AI, ensuring flexibility and adaptability to evolving AI practices and methodologies.
- Flexibility and control of communication flow: The artifact should provide a flexible and configurable framework for defining and controlling communication flows between agents.
- Distributed agents orchestration: The system must support orchestrating distributed agents across multiple environments, enabling them to execute tasks in a coordinated manner and collaborate effectively within a networked ecosystem.
- Specification and control of agent execution order: It should provide a mechanism to define, enforce, and monitor the execution order of agents to ensure task prioritization, synchronization, and compliance with operational workflows.
- Support for agent hierarchies: The architecture must allow for hierarchical structuring of agents, enabling an agent to be part of other agents, with the ability to delegate tasks and responsibilities across different levels of the hierarchy [31].
- Ease of integration into existing systems: The artifact should support effortless integration with existing software systems and infrastructure, minimizing the need for extensive modifications or reconfigurations.
- Support for agents using different tech and protocols: The artifact must facilitate the use of agents developed in different programming languages and support diverse communication interfaces (e.g., Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), etc.), promoting flexibility and inclusivity.
- Ease of component replacement: Components, including agents and communication interfaces, should be modular and replaceable without significant effort, enabling users to upgrade or replace parts of the system as technology evolves.

- Operational scalability: The system should demonstrate the ability to scale reasonably well, maintaining acceptable performance and responsiveness as the number of agents or the complexity of their interactions increases.

- Resilience and fault tolerance: The artifact should include mechanisms for handling agent failures, ensuring uninterrupted operation and graceful degradation when issues arise.

- Support for cloud computing: The artifact should be designed to leverage cloud computing environments, ensuring access to distributed resources, high availability, and scalability while reducing infrastructure overhead for users.

4.3 Proposal

For clarity, the proposed solution is divided into two components, each considered separately in relation to the research objectives:

- A programming language for orchestrating heterogeneous microservices in Multi-agent systems (MASs) architecture

- A declarative engine for controlling communication flows between intelligent agents

4.3.1 Programming language

The proposed programming language is based on the π-Calculus [76]. The rationale for adopting π-Calculus as the formal foundation of this language is its suitability for modeling systems that emphasize interaction, communication, and synchronization between dynamic components [76]. It focuses on defining the communication behavior between agents and other entities in the system. In this proposal, agents do not communicate with each other directly. Instead, all interactions are mediated through channels and environments, which serve as intermediaries for sending and receiving messages.

The language enables the definition of communication flows in a simple yet precise way. It also allows for message transformation between different formats and supports specifying the protocol to be used for each interaction. The grammar and parser for the language will be implemented using ANother Tool for Language Recognition (ANTLR) [75].

The development of the proposed programming language addresses the first research objective.

4.3.1.1 Elements

The proposal is that the programming language consists of five core elements around which its features are built. These elements provide the foundation for expressing orchestration logic in a structured and modular way. The elements are as follows:

- Agent: Represents a microservice that encapsulates business logic and performs execution. Agents communicate indirectly through channels and environments rather than directly with one another.
- Channel: An artifact-specific component that receives messages from agents and forwards them to other subscribed agents.
- Holon: A semantically grouped cluster of agents that can interact with each other through a shared environment [31].
- Environment: A communication interface that enables interaction between holons. An environment can be defined at the holon's input, output, or both.
- Execution flow: Defines the order and dependencies for initializing and coordinating agent execution.

The reserved keywords used to define these elements in the programming language are `agent`, `channel`, and `environment`. For integrating external holons [31], the proposed keyword is `import`, followed by the name of the holon. A conceptual example is shown below:

```
1 import <holon name>
```

Listing 4.1: Conceptual holon import

The design of these core elements, along with the semantics and pragmatic usage patterns described in the following sections, directly addresses the second research question on shaping the syntax, semantics, and pragmatics of a programming language for the orchestration of agents.

4.3.1.2 Communication

Communication, specifically the specification of communication flows, is a crucial aspect of the proposed artifact and directly addresses the first research question regarding the types of communication flows the programming language should support in the context of modern domains. Among the defined language constructs, the elements directly involved in communication are agent, channel, and environment. Each communication flow can be configured by specifying the direction of communication, the protocol used, and any transformation required from the input format to the output format.

The communication flow specification defines how an agent communicates with a channel, environment, or standard input-output interface, and in what direction. The proposal is that the left-hand side of a communication flow statement represents the sender, followed by the communication protocol, and then the receiving entity on the right-hand side.

An agent should be implemented to communicate with one or more of the following:

- Channel: By specifying the name of the channel to which the agent sends or from which it receives messages.
- Environment: By indicating the environment input using the keyword `ENV_INPUT`, and the environment output using `ENV_OUTPUT`.
- Standard input-output interfaces:
  - `STDIN`
  - `STDOUT`
  - `STDERR`

Although channels and environments participate in the message routing process, the primary point for defining communication flows is at the agent level. To allow agents to refer to themselves within specifications, the reserved keyword `self` is introduced.

```
1 <sending entity> <communication protocol> <receiving entity>
```

Listing 4.2: Conceptual agent communication flow specification

Based on the analysis of existing literature on agent communication, and recognizing that the communication in this context is primarily internal and focused on orchestration, the proposal is to support two core protocols that can be configured within the orchestration specification:

- TCP, represented in the language as ->
- User Datagram Protocol (UDP), represented as *->

These symbolic annotations allow protocol selection to be embedded directly within flow definitions. For instance, the following example demonstrates the use of the UDP protocol:

```
1 <sending entity> *-> <receiving entity>
```

Listing 4.3: Conceptual protocol specification using UDP

Channels and environments should be able to transform messages as they pass through to subscribed agents. That means that an input message reaching either of these entities can be transformed into a different output before being sent out. The proposal is to use templates containing variables that describe which information should be preserved during format transformation. A variable is defined using a question mark (`?`) followed by the variable name. To allow clearer transformations between standard formats, each message format should be encapsulated within a function call.

The supported message transformations are:

- JavaScript Object Notation (JSON)
- Extensible Markup Language (XML)
- Regular Expression (REGEX)

Below is an example of an input message template in JSON format, with a variable `?var`:

```
1 json({"data": ?var})
```

Listing 4.4: Conceptual input message template with variable

Similarly, below is an example of an output message template in XML format, using the extracted variable `?var`:

```
1 xml(<Abc test="?var" />)
```

Listing 4.5: Conceptual output message template with variable

If a channel or environment is not meant to perform any message transformation (i.e., it is transparent), the notation should begin with the entity keyword, followed by the name (for channels), and end with a period, as shown below:

```
1 <entity> <name>? .
```

Listing 4.6: Conceptual transparent channel or environment

In the case of a transformative channel, the definition becomes multiline. The second line defines the input and output templates, along with the selected protocol. For example:

```
1 channel <name> :
2     <input template> <communication protocol> <output template>
```

Listing 4.7: Conceptual transformative channel

Since a holon may have both input and output environments, the transformation in this case is defined using separate input and output declarations. Each line starts with the keyword `input` or `output`, followed by the directed protocol, and ends with the corresponding template. An example is given below:

```
environment :
    input <directed communication protocol> <input template>
    output <directed communication protocol> <output template>
```

Listing 4.8: Conceptual transformative environment

4.3.1.3 Execution flows

The main purpose of execution flows is to define the order in which agents should be started. These flows can also include conditional logic. This information is used during orchestration to determine which agents should be started and when.

An execution flow begins with the `start` keyword, followed by agent names and operators that define sequencing or conditions between them. A conceptual example of an execution flow with a single agent is shown below:

```
start <agent name> <operator>?
```

Listing 4.9: Conceptual execution flow specification

The proposal also supports the specification of more complex execution flows involving multiple agents and conditional dependencies. Below is a list of proposed execution flows to be supported by the orchestration platform:

- Sequential: Agents are started one after another. As soon as the preceding agent completes, the next one is started. This is the default behavior and does not require an operator.
- Parallel: Agents are started simultaneously. This is specified using the operator `|`.
- On failure: An agent is started only if the preceding agent fails. This is specified using the operator `!`.
- On success: An agent is started only if the preceding agent completes successfully. This is specified using the operator `&`.
- Restart: The agent should automatically restart as soon as it completes, regardless of success or failure. This is specified using the operator `+`.

4.3.2 Declarative engine

The objective of the declarative engine is to consume agents' orchestration specifications written in the proposed programming language and orchestrate the execution of agents accordingly. This includes managing their communication, tracking their life cycles, and coordinating their execution in the required order. The declarative engine should also be capable of recursively loading and starting other holons along with their respective agents if defined in the specification.

To support this, the proposal includes building a declarative engine and the associated Python-based orchestration platform. This platform will serve as centralized middleware that connects agents and provides the necessary infrastructure to integrate both existing and new agents designed with the platform in mind. Agents should be initialized through the orchestration platform and should primarily communicate with it. The orchestration platform will then relay messages to the appropriate agents. Therefore, each agent must have a corresponding agent definition file (`.ad`) that describes its communication interface, allowing the orchestration platform to correctly interact with it.

The orchestration platform will also be responsible for managing environments and channels, entities specific to the platform and abstracted away from the end user implementation. Engineers integrating their MASs should not be burdened with managing these components directly. Given that the platform supports cloud-native operations, agents can be started as UNIX processes [29], Docker containers [110], or pods within a Kubernetes cluster [18].

Developing the declarative engine and the orchestration platform aligns with the second research objective.

4.3.2.1 Agent integration

This artifact should integrate into existing MASs with minimal modification. Since the orchestration platform handles all communication with agents, it must be aware of how to interact with each agent, including its public interface, supported protocols, data formats, and communication modes (e.g., one off messaging, streaming, etc.).

Each agent should have a definition file written in Yet Another Markup Language (YAML) that includes the following fields:

- Name: A unique name for the agent.
- Description: A description of the agent's purpose.
- Start command: Instructions for launching the agent.
- Start up mode: Specifies how the agent should be started (e.g., UNIX process, Docker, or Kubernetes) [29, 110, 18].

- Communication interface - Defines the communication interface of the agent, including:
  - The protocol used for communication (e.g., HTTP, WebSocket (WS), Extensible Messaging and Presence Protocol (XMPP)) [93].
  - The expected message format (e.g., JSON, XML, plain text).
  - Message type, which specifies whether the agent uses one-off messaging or streaming.
  - Delimiter and end indicator used to segment or terminate streamed messages when applicable.

In the initial implementation, the proposed orchestration platform will support a predefined set of communication protocols that combine both widely adopted standards and lightweight, custom-built solutions. The proposed protocols are:

- `STDIN`: For direct interaction with local UNIX processes [29].
- `FILE`: For batch-style input/output via the file system.
- HTTP: For standard request-response communication over the web.
- WS: For real-time, bidirectional communication using WS.
- Netcat: For lightweight socket-based messaging using TCP and UDP.

For each agent, the orchestration platform should spawn an agent wrapper, which is a runtime component that mediates between the agent and the platform. The wrapper should implement a communication socket that can adapt incoming and outgoing messages across various protocols and data formats to the unified communication model used within the orchestration platform.

4.3.2.2 Holons and communication management

The orchestration platform is responsible for managing and exchanging metadata such as life cycle events with agents, environments, channels, and holons using the XMPP protocol, incorporating the Smart Python Agent Development Environment (SPADE) framework [93, 70, 31].

Each holon should maintain an address book containing the XMPP addresses of all agents, environments, and channels within its cluster. If necessary, it may also include addresses for external holons. This address book is critical for enabling communication across distributed components in the system [93, 70]. By organizing holons with their own communication flows and enabling coordination between holons through known addresses, this approach helps address the third research question on how to support the design process of complex method ensembles using holonic systems [31].

4.3.2.3 Agent life cycle events

Tracking agent life cycle events is essential for enabling more complex execution flows, such as conditional starts and failure recovery. It also facilitates a more fault-tolerant system capable of notifying other entities in the ecosystem about agent-specific failures.

The following life cycle events should be tracked:

- Startup: The agent has been launched.
- Readiness: The agent signals it is ready to process tasks.
- Completion: The agent terminates, either successfully or due to failure.

This should be achieved through proactive communication among all entities within the ecosystem, ensuring that critical state changes such as agent readiness, termination, or failure are reliably propagated to relevant components.

4.3.2.4 Orchestration

Using the orchestration specification and visibility into each agent's life cycle, the platform can coordinate agent execution according to the defined execution flow. It must ensure proper sequencing, respect agent dependencies, and satisfy communication protocol and configuration constraints. Orchestration must be performed in a synchronized manner, ensuring consistency across agent interactions and maintaining correctness in distributed settings.

4.3.2.5 Cloud support

The orchestration platform shall offer native support for cloud deployments. Since the platform is responsible for agent startup, it can control the deployment method. The following execution environments are proposed:

- Docker — Agents can be run in Docker containers and deployed using Docker Swarm [110].
- Kubernetes — Agents can be deployed as pods managed within a Kubernetes cluster [18].
- UNIX process - For simple or lightweight deployments [29].

This flexibility enables integration into a variety of deployment scenarios, from research testbeds to production-grade systems.

Chapter 5

Implementation

Based on the collected requirements and the resulting proposal, an artifact named `APi` has been developed [78]. The primary goal of this artifact is to enable enhanced agent orchestration capabilities through a programming language for specifying communication flows, along with the corresponding orchestration platform.

In essence, the platform acts as both a coordinator and a facilitator, ensuring smooth and efficient communication between agents while enforcing structured interactions. At the core of this orchestration lies the communication flow specification, also referred to as the agents' orchestration specification, which is written in the proposed programming language. This language is designed to offer high flexibility in defining communication dynamics, covering essential aspects such as message direction, protocol, message type, and other characteristics required for structured agent collaboration.

Once the communication flow specification is formulated, the declarative engine and the orchestration platform interpret it and use it as a guide to coordinate and orchestrate agents. This process ensures that agents follow predefined communication patterns without requiring direct peer-to-peer interactions. Instead, each agent interfaces with the orchestration platform, which determines the appropriate routing and message delivery according to the specified communication flows. This approach simplifies integration while enforcing modularity and separation of concerns, making the system more scalable and adaptable to various environments.

A fundamental feature of the artifact is the independence of agents. The platform is designed to accommodate agents developed in any technology, provided they adhere to the required communication interface specifications. This allows developers to design and implement agents using different frameworks, languages, or infrastructures, as long as they can communicate with the orchestration platform in a standardized way. To integrate a new agent into the MAS, an agent definition must be defined, outlining the communication contract that governs its interactions with the platform. This specification ensures compatibility, allowing the orchestrator to mediate messages between agents without requiring direct communication between them.

By centralizing communication through the orchestration platform, the system gains a high degree of control, traceability, and flexibility. Since agents do not establish direct connections, their interactions remain decoupled, reducing dependencies and making it easier to modify or replace individual agents without disrupting the overall system. The platform thus provides a robust foundation for designing complex multi-agent environments, where autonomous components collaborate under well-defined communication rules.

Figure 5.1 illustrates the orchestration process, showing how the agents' orchestration specification is consumed by the orchestration platform, how coordination occurs, and how messages are routed to the appropriate recipients, i.e., the agents. The socket and agent definition components are responsible for bridging differences in communication configurations.

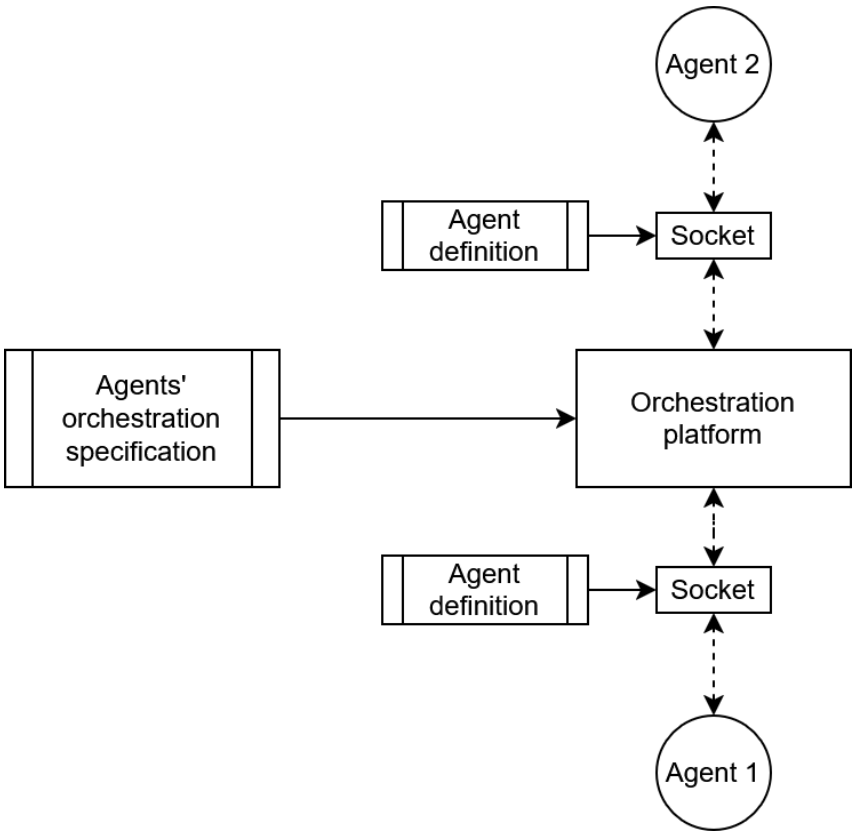

Figure 5.1: Orchestration platform communication with agents

The following implementation description is divided into two high level components, namely the programming language and the orchestration platform, which also serves as the declarative engine.

5.1 Programming language

The programming language is based on the π-Calculus [76], a formal model designed for describing and analyzing concurrent systems. Its primary role is to enable engineers to specify the communication between different types of elements involved in orchestration, specifically holons, channels, agents, and environments.

Unlike imperative programming languages that focus on sequential execution, π-Calculus allows for the representation of processes that evolve and interact asynchronously. This makes it particularly well-suited for defining communication protocols, distributed systems, and multi-agent interactions, where entities operate independently yet must coordinate effectively.

The primary objective of this language is to provide an intuitive and user-friendly way to specify how agents communicate. Users define agents' orchestration specification using `.api` files, which serve as structured specifications written in the programming language. These specification describe how different elements exchange information, ensuring clear and enforceable communication patterns.

To facilitate parsing and validation, the grammar of the language is defined using ANTLR, a widely used parser generator. ANTLR enables the compilation of the language, generating the necessary lexer and parser components to [75]:

- Validate whether a given `.api` file adheres to the syntactic rules.
- Convert the high-level communication constructs into a structured representation suitable for execution by the declarative engine.

Since the declarative engine is implemented in Python, the artifacts generated by ANTLR, including the lexer and parser, are also designed to be compatible with Python. This ensures integration between the language specification and its execution environment.

This programming language introduces a declarative approach to specifying communication in MASs, enabling:

- Clear separation of concerns between agents, their communication channels, and the environments in which they operate.
- A formal yet practical specification language that allows engineers to define interactions without directly handling lower-level concurrency primitives.
- Automatic validation and transformation of communication specifications into executable representations through ANTLR.

5.1.1 Semantic analysis

The programming language is formally specified using ANTLR, with its syntax and semantics defined across three distinct grammar files (`.g4` extension). These grammar files are:

1. JSON grammar: Defines constructs related to JSON-based data representation.
2. XML grammar: Governs XML-based structures and their integration within the language.
3. `APi` grammar: Serves as the core foundation of the language, defining key syntactic constructs and operational semantics.

The following JSON grammar is adapted from ANTLR Reference [72] and slightly modified for specialized use. It handles fundamental JSON constructs such as objects, arrays, values (including strings, numbers, booleans, and null), and also introduces additional tokens such as `VARIABLE` and `SPEC_CHAR` for more flexible syntax. Objects are enclosed by braces and contain key-value pairs separated by commas, arrays are enclosed by square brackets and contain comma-separated values, and values can include nested objects or arrays. The grammar also defines a custom `VARIABLE` token and includes mechanisms for recognizing Unicode escape sequences, ensuring robust string handling.

```
grammar JSON;

json
   : value
   ;

obj
   : '{' pair (',' pair)* '}'
   | '{' '}'
   ;

pair
   : STRING ':' value
   | VARIABLE ':' value
   ;

arr
   : '[' value (',' value)* ']'
   | '[' ']'
   ;

value
   : VARIABLE
```

```
24      | STRING
25      | NUMBER
26      | SPEC_CHAR
27      | obj
28      | arr
29      | 'true'
30      | 'false'
31      | 'null'
32      ; // TODO: find out why SPEC_CHAR is here
33
34  SPEC_CHAR : ~('a'..'z' | 'A' .. 'Z' | '0' .. '9' | ':' | '.' | '-' | '>'
        | '<' | '/' | ' ' ) ;
35
36  VARIABLE : '?' IDENT ;
37
38  IDENT   : NameStartChar1 NameChar1* ; //[a-zA-Z_] [a-zA-Z0-9]*;
39
40  fragment
41  NameChar1     :   NameStartChar1
42              |   '-' | '_' | INT
43              |   '\u00B7'
44              |   '\u0300'..'\u036F'
45              |   '\u203F'..'\u2040'
46              ;
47
48  fragment
49  NameStartChar1
50              :   [a-zA-Z]
51              |   '\u2070'..'\u218F'
52              |   '\u2C00'..'\u2FEF'
53              |   '\u3001'..'\uD7FF'
54              |   '\uF900'..'\uFDCF'
55              |   '\uFDF0'..'\uFFFD'
56              ;
57
58  STRING
59      : '"' (ESC1 | SAFECODEPOINT1)* '"'
60      | '\'' (ESC2 | SAFECODEPOINT2)* '\''
61      ;
62
63  fragment ESC1
64      : '\\' (["\\/bfnrt] | UNICODE)
65      ;
66
67  fragment ESC2
68      : '\\' (['\\/bfnrt] | UNICODE)
69      ;
```

```
70
71  fragment UNICODE
72      : 'u' HEX HEX HEX HEX
73      ;
74
75  fragment HEX
76      : [0-9a-fA-F]
77      ;
78
79  fragment SAFECODEPOINT1
80      : ~ ["\\\u0000-\u001F]
81      ;
82
83  fragment SAFECODEPOINT2
84      : ~ ['\\\u0000-\u001F]
85      ;
86
87  NUMBER
88      : '-'? INT ('.' [0-9] +)? EXP?
89      ;
90
91  fragment INT
92      : '0' | [1-9] [0-9]*
93      ;
94
95  fragment EXP
96      : [Ee] [+\-]? INT
97      ;
98
99  // \- since - means "range" inside [...]
100
101 SPACE
102     : [ \r]+ -> skip
103     ;
```

Listing 5.1: JSON grammar

The XML lexer obtained from ANTLR public examples [73] demonstrates how ANTLR modes are used to handle content outside and inside XML tags, as well as processing instructions. It recognizes tags, attributes, comments, `CDATA` sections, and `DTD` declarations, among other XML constructs. The lexer transitions into different modes upon encountering certain tokens (like `<` or `<?`), enabling it to properly tokenize elements and attributes within XML documents. It also includes rules for entity references, character references, and names that allow letters, digits, underscores, hyphens, and specific Unicode ranges.

```
1  lexer grammar XMLLexer;
2
```

```
3  COMMENTXML  :   '<!--' .*? '-->' ;
4  CDATA       :   '<![CDATA[' .*? ']]>' ;
5
6  DTD         :   '<!' .*? '>'            -> skip ;
7  EntityRef   :   '&' IDENT ';' ;
8  CharRef     :   '&#' DIGIT+ ';'
9              |   '&#x' HEXDIGIT+ ';'
10             ;
11 SEA_WS      :   (' '|'\t'|'\r'? '\n')+ ;
12
13 OPEN        :   '<'                     -> pushMode(INSIDE) ;
14 XMLDeclOpen :   '<?xml' S               -> pushMode(INSIDE) ;
15 SPECIAL_OPEN:   '<?' IDENT               -> more, pushMode(PROC_INSTR) ;
16
17 TEXT        :   ~[<&]+ ;        // match any 16 bit char other than <
       and &
18
19 // ----------------- Everything INSIDE of a tag ---------------------
20 mode INSIDE;
21
22 CLOSE       :   '>'                     -> popMode ;
23 SPECIAL_CLOSE:  '?>'                    -> popMode ; // close <?xml...?>
24 SLASH_CLOSE :   '/>'                    -> popMode ;
25 SLASH       :   '/' ;
26 EQUALS      :   '=' ;
27 STRINGXML   :   '"' ~[<"]* '"'
28             |   '\'' ~[<']* '\''
29             ;
30 Name        :   NameStartChar NameChar* ;
31 S           :   [ \t\r\n]               -> skip ;
32
33 fragment
34 HEXDIGIT    :   [a-fA-F0-9] ;
35
36 fragment
37 DIGIT       :   [0-9] ;
38
39 fragment
40 NameChar    :   NameStartChar
41             |   '-' | '_' | '.' | DIGIT
42             |   '\u00B7'
43             |   '\u0300'..'\u036F'
44             |   '\u203F'..'\u2040'
45             ;
46
47 fragment
48 NameStartChar
```

```
49              :   [:a-zA-Z]
50              |   '\u2070'..'\u218F'
51              |   '\u2C00'..'\u2FEF'
52              |   '\u3001'..'\uD7FF'
53              |   '\uF900'..'\uFDCF'
54              |   '\uFDF0'..'\uFFFD'
55              ;
56
57  IDENT   : NameStartChar1 NameChar1* ; //[a-zA-Z_] [a-zA-Z0-9]*;
58
59  fragment
60  NameChar1    :   NameStartChar1
61              |   '-' | '_' | INT
62              |   '\u00B7'
63              |   '\u0300'..'\u036F'
64              |   '\u203F'..'\u2040'
65              ;
66
67  fragment
68  NameStartChar1
69              :   [a-zA-Z]
70              |   '\u2070'..'\u218F'
71              |   '\u2C00'..'\u2FEF'
72              |   '\u3001'..'\uD7FF'
73              |   '\uF900'..'\uFDCF'
74              |   '\uFDF0'..'\uFFFD'
75              ;
76
77  fragment INT
78     : '0' | [1-9] [0-9]*
79     ;
80
81  // ----------------- Handle <? ... ?> ----------------------
82  mode PROC_INSTR;
83
84  PI          :   '?>'                    -> popMode ; // close <?...?>
85  IGNORE      :   .                       -> more ;
```

Listing 5.2: XML grammar

The `APi` grammar establishes the fundamental tokens and parsing rules that dictate the structure and behavior of programs written in this language. The lexical rules that define the tokens in the `APi` grammar are outlined below.

One of the key capabilities of the language is handling REGEXs within its execution model. The `s_regex` construct allows the definition of a regular expression-based operation, where a REGEX pattern is applied to a given string. Additionally, the language

provides explicit tokens for defining input and output operations, making it easier to handle data flow.

- `s_regex`: A function-like construct that applies a REGEX to a given `STRING` enclosed in parentheses.
- `INPUT`: Defines an input directive.
- `OUTPUT`: Specifies an output directive.

To facilitate interaction with system input-output, the language includes predefined tokens for managing standard input, output, and error streams. These tokens provide a structured mechanism to handle process communication efficiently.

- `STDOUT`: Represents the standard output stream.
- `STDERR`: Defines the standard error stream.
- `STDIN`: Refers to the standard input stream.

The language includes several core keywords that define fundamental constructs, such as types, module imports, and execution environments. These tokens shape the semantics of function execution and modularity.

- `VOID`: Represents a void return type, indicating the absence of a meaningful value.
- `IMPORT`: Declares an import statement, enabling the inclusion of external holon.

Control flow in the language is designed to be event-driven and resilient, allowing the specification of success and failure handlers, as well as mechanisms for parallel execution and restart logic.

- `ONSUCCESS (&)`: Defines a directive or operator that executes upon success of a preceding operation.
- `ONFAIL (!)`: Specifies an operation that executes in case of failure.
- `RESTART (+)`: Represents a restart mechanism, possibly allowing retry logic for execution.
- `PARALLEL (|)`: Denotes parallel execution.

The language also defines agent-based constructs, used for distributed and concurrent execution. Agents interact through channels and environments and are capable of self-referential behavior, allowing dynamic and autonomous execution.

- `START`: Marks the beginning of an execution flow.
- `SELF`: Represents a self-referential element.
- `AGENT`: Defines an agent element.
- `CHANNEL`: Represents channel element.
- `ENVIRONMENT`: Represents environment element.

Since structured data handling is an essential feature, the language supports multiple data formats, particularly JSON and XML, which allow integration with external systems.

- REGEX: Defines REGEX data handling.
- JSON: Defines JSON data handling.
- XML: Specifies XML data handling.

The language supports two forms of inter-process communication, TCP and UDP.

- `A_SENDS ('->')`: Defines a basic send operation, likely between agents or processes.
- `C_SENDS (TCP | UDP)`: Specifies connection-oriented (TCP) or connectionless (UDP) message transmission.
- `E_SENDS (TCP_BW | UDP_BW)`: Denotes bidirectional communication over TCP or UDP.
- `TCP ('->')` and `UDP ('*->')`: Represent unidirectional transmission from left to right using TCP and UDP.
- `TCP_BW ('<-')` and `UDP_BW ('<-*')`: Represent unidirectional transmission from right to left using TCP and UDP.

The language also includes special tokens to handle null values, comments, and formatting. These tokens ensure structured parsing and readability.

- `NIL ('0')`: Represents a null or empty value, serving as a placeholder for undefined data.
- `COMMENT1 ('// ...')` and `COMMENT2 ('/* ... */')`: Define single-line and multi-line comments, allowing annotation and documentation within code.
- `WS`: Handles whitespace characters, ensuring proper tokenization.
- `NEWLINE`: Recognizes line breaks, facilitating structured parsing.

- `TAB`: Defines tab characters, supporting indentation-based formatting if applicable.

Below is the definition of lexer tokens for the `APi` grammar.

```
s_input : s_json | s_xml | s_regex ;
s_output : s_json | s_xml ;
s_xml : XML '(' xml ')';
s_json : JSON '(' json ')';
s_regex : REGEX '(' STRING ')';
INPUT : 'input' ;
OUTPUT : 'output' ;
STDOUT : 'stdout' ;
STDERR : 'stderr' ;
STDIN : 'stdin' ;
VOID : 'void' ;
IMPORT : 'import' ;
ENVIRONMENT : 'environment' ;
ONSUCCESS : '&' ;
ONFAIL : '!' ;
RESTART : '+' ;
PARALLEL : '|' ;
START   : 'start' ;
SELF    : 'self' ;
AGENT   : 'agent' ;
CHANNEL : 'channel' ;
REGEX : 'regex' ;
JSON : 'json' ;
XML : 'xml' ;
A_SENDS  : '->' ;
C_SENDS : TCP | UDP ;
E_SENDS : TCP_BW | UDP_BW ;
TCP : '-->' ;
UDP : '*->' ;
TCP_BW : '<--' ;
UDP_BW : '<-*' ;
NIL     : '0' ;
COMMENT : COMMENT1 | COMMENT2 ;
COMMENT1: '//'~[\n]* ;
COMMENT2: '/*' .*? '*/' ;
WS : [ \f\r] ;
NEWLINE : [\n] ;
TAB : [\t] ;
```

Listing 5.3: `APi` grammar lexer tokens

The `APi` grammar defines the core execution flow of the language, specifying how programs are structured and how different components interact. The entry point of a program is represented by `api_program`, which consists of multiple elements such as

imports, environment definitions, channels, agents, and execution rules.

`api_program` represents the top-level structure of the program. It can include `s_import` statements for external modules (specifically, holons), environment definitions (`s_environment` and `s_environment_forward`), channel specifications (`s_channel` and `s_channel_forward`), agent definitions (`s_agent`), a start construct (`s_start`) for execution flows, and also allows comments (`COMMENT`) or newlines (`NEWLINE`).

The environment is introduced using the `environment` keyword, which allows defining or forwarding environment configurations. These environments specify how data flows into and out of the program.

- `s_environment`: Declares an execution environment with multiple input (`iflow`) or output (`oflow`) flows.
- `s_environment_forward`: Forwards or references an existing environment definition elsewhere.
- `iflow`: Defines how an external input (`INPUT`) is sent into the system (`C_SENDS`) and bound to a specific input spec (`s_input`).
- `oflow`: Describes how an output (`OUTPUT`) is transmitted out of the system (`E_SENDS`), paired with a particular output spec (`s_output`).

The `s_start` rule indicates where an execution flow begins, linking to a `pi_expr`. This process expression describes the flow of execution, allowing parallelism, success and failure handlers, and sequential logic.

- `s_start`: Initiates execution by referencing a process expression (`pi_expr`).
- `pi_expr`: Allows grouping expressions with parentheses, chaining operations in parallel (`PARALLEL`), reacting to success (`ONSUCCESS`) or failure (`ONFAIL`), restarting execution (`RESTART`), and defining sequential or functional calls (`IDENT ( arglist )?`).

Agents represent autonomous entities in the language. Each agent has a name, optional arguments, and rules (`aflow`) describing how it sends messages between channels.

- `s_agent`: Defines an agent with a unique identifier, optional arguments, and communication logic.
- `arglist`: Lists arguments passed to the agent, enabling parameterization.
- `aflow`: Specifies message-passing behavior (`A_SENDS`) between channels (or special identifiers like `SELF`, `NIL`, `STDIN`, `STDOUT`, `STDERR`, `VOID`).

Channels are communication interfaces that can either be used as pass-through mechanisms or configured to apply transformations to the transmitted data.

- `s_channel`: Declares a channel, assigning it an identifier and specifying its behavior.
- `s_channel_forward`: References a channel by name and forwards its definition.
- `s_channel_spec`: Provides the mechanics of a channel by specifying `s_input`, how it is sent (`C_SENDS`), and the corresponding `s_output`.

Import statements allow bringing external modules or dependencies into the program scope, enabling modular design.

- `s_import`: Integrates an external holon via `IMPORT IDENT`.

By combining these rules, the grammar supports a robust structure for programs that handle environments, agents, channels, and execution flows. The process expressions (`pi_expr`) provide flexibility for parallel execution, error handling, and sequential composition, while agents and channels foster asynchronous communication.

```
1  api_program : ( s_import | s_environment | s_environment_forward |
       s_channel | s_channel_forward | s_agent | s_start | COMMENT | NEWLINE
        )*? ;
2  s_environment : ENVIRONMENT WS ':' NEWLINE ( iflow | oflow )+ ;
3  s_environment_forward : ENVIRONMENT WS '.' NEWLINE ;
4  iflow : TAB INPUT WS C_SENDS WS s_input NEWLINE ;
5  oflow : TAB OUTPUT WS E_SENDS WS s_output NEWLINE ;
6  s_start : START WS pi_expr ;
7  pi_expr : '(' pi_expr ')'
8          | pi_expr RESTART
9          | pi_expr WS PARALLEL WS pi_expr
10         | pi_expr WS ONSUCCESS WS pi_expr
11         | pi_expr WS ONFAIL WS pi_expr
12         | pi_expr WS pi_expr
13         | IDENT ( arglist )?;
14 s_agent : AGENT WS IDENT ( arglist WS )? ':' NEWLINE aflow+ ;
15 arglist : '(' IDENT (WS IDENT)* ')';
16 aflow : TAB valid_channel WS A_SENDS WS valid_channel NEWLINE ;
17 valid_channel : IDENT | SELF | NIL | STDIN | STDOUT | STDERR | VOID ;
18 s_channel : CHANNEL WS IDENT WS ':' NEWLINE s_channel_spec ;
19 s_channel_forward : CHANNEL WS IDENT WS '.' NEWLINE ;
20 s_channel_spec : TAB s_input WS C_SENDS WS s_output NEWLINE;
21 s_import : IMPORT IDENT NEWLINE ;
```

Listing 5.4: `APi` grammar lexer rules

5.1.2 Parser

The parser is generated from an ANTLR grammar written to specify the syntax of the `APi` language. Its role is to read the code written in `APi`, recognize the constructs of agents, channels, environments, execution plans, and holons, and produce internal Python data structures that the declarative engine later consumes. When an `APi` agents' orchestration specification is parsed, it is processed according to the rules defined in the grammar file. From these rules, ANTLR generates both a lexer and a parser. The lexer converts raw text into a series of tokens, and the parser arranges those tokens into a parse tree. After that, a listener class is used to walk this parse tree and perform custom logic at each recognized rule. In this system, the listener in question is a Python class named `APi`, which inherits from the automatically generated `APiListener`.

One of the core components used by the listener is the `APiNamespace` class. This class acts as a container for the objects the parser discovers in the code, such as agents, channels, environments, and holons. It also stores any execution plans that the `APi` specification includes. The code for `APiNamespace` is:

```
class APiNamespace:
    def __init__(self):
        self.agents = []
        self.channels = []
        self.environment = None
        self.execution_plans = []
        self.holons = []

    def add_agent(self, agent):
        self.agents.append(agent)

    def add_channel(self, channel):
        self.channels.append(channel)

    def add_environment(self, environment):
        self.environment = environment

    def add_execution_plan(self, plan):
        self.execution_plans.append(plan)

    def add_holon(self, holon):
        self.holons.append(holon)
```

Listing 5.5: `APiNamespace` implementation

Each list or attribute in `APiNamespace` corresponds to a top-level construct in the language. `agents` holds all the agent definitions, `channels` holds the channel definitions, `environment` stores a single environment object that captures input and output para-

meters, `execution_plans` is a list of any instructions about how to start or coordinate the specification, and `holons` collects references to any externally imported modules or components.

The listener class is where parsing logic is actually implemented. It overrides the generated methods from `APiListener` so that it can collect information about the language constructs as the parse tree is walked. This is done by populating the `APiNamespace` object. The `APi` class defines its own init method to instantiate an `APiNamespace` and maintain a stack used for intermediate parse results. The stack is a Python list named `STACK`.

Whenever a rule in the grammar begins to match parts of the `APi` code, ANTLR calls the enter method for that rule in the listener. In contrast, when the rule finishes, it calls the exit method. It is typical in this design to do most of the logic in the exit methods. For instance, take the rule for parsing the environment. The environment can be specified with input flows, output flows, or both, using symbolic arrow notations that correspond to different protocols. The listener records these flows in the `STACK` while the parser is within the `iflow` or `oflow` rules. Then, when the environment rule finishes, its exit method assembles and adds the environment dictionary to the `APiNamespace`:

```
def exitS_environment(self, ctx: APiParser.S_environmentContext):
    input = None
    input_protocol = "tcp"
    output = None
    output_protocol = "tcp"
    while len(self.STACK) > 0:
        entry = self.STACK.pop()
        if entry["type"] == "input":
            input = entry["value"]
            input_protocol = entry["protocol"]
        elif entry["type"] == "output":
            output = entry["value"]
            output_protocol = entry["protocol"]
    environment = {
        "input": input,
        "output": output,
        "input_protocol": input_protocol,
        "output_protocol": output_protocol,
    }
    self.ns.add_environment(environment)
```

Listing 5.6: `APi` grammar transformative environment rule parsing implementation

This logic gathers whatever was put on the stack in the form of input or output flows. The dictionary it constructs collects the input, output, and their respective protocols,

defaulting to TCP when the user does not specify otherwise. Finally, the dictionary is added to `self.ns`, which is the `APiNamespace` instance. A similar approach is taken for an environment that is only declared in a forward manner, meaning no explicit input or output are defined:

```
def exitS_environment_forward(self, ctx: APiParser.
    S_environment_forwardContext):
    environment = {
        "input": None,
        "output": None,
        "input_protocol": "tcp",
        "output_protocol": "tcp",
    }
    self.ns.add_environment(environment)
```

Listing 5.7: `APi` grammar forward environment rule parsing implementation

`APi` code can include a variety of flows such us `iflow` and `oflow`. Their enter methods push entries onto the stack so that the environment or agent rule that encloses them can gather them later:

```
def enterIflow(self, ctx: APiParser.IflowContext):
    protocol_symbol = ctx.children[3].getText()
    protocol = "tcp" if protocol_symbol == "-->" else "udp"
    input = ctx.children[5].getText()
    self.STACK.append({"type": "input", "value": input, "protocol":
        protocol})

def enterOflow(self, ctx: APiParser.OflowContext):
    protocol_symbol = ctx.children[3].getText()
    protocol = "tcp" if protocol_symbol == "<--" else "udp"
    output = ctx.children[5].getText()
    self.STACK.append({"type": "output", "value": output, "protocol":
        protocol})
```

Listing 5.8: `APi` grammar `iflow` rule parsing implementation

Agents rely on a similar pattern. The grammar rule for an agent might accept a name, an optional argument list, and multiple flows that define communications or relationships. The exit method for `s_agent` looks like this:

```
def exitS_agent(self, ctx: APiParser.S_agentContext):
    a_name = ctx.children[2].getText()
    flows = []
    args = None
    while len(self.STACK):
```

```
6           exp = self.STACK.pop()
7           type = exp["type"]
8           value = exp["value"]
9           if type == "flow":
10              flows.append(value)
11          else:
12              args = value
13      agent = {"name": a_name, "args": args, "flows": flows}
14      self.ns.add_agent(agent)
```

Listing 5.9: `APi` grammar `agent` rule parsing implementation

This series of parsing steps and listener methods transforms an `APi` program text into structured data within the `APiNamespace`, which is then consumed by the declarative engine. ANTLR builds the parse tree, while the listener extracts and assembles components using a stack-based approach. As each rule is exited, stored sub-components are retrieved and combined into cohesive entities. This method ensures a clear separation between parsing logic and data representation, making the parser maintainable and adaptable as the language evolves. Similar steps are followed for other entities, ensuring consistency in parsing across different constructs.

5.1.3 Agents' orchestration specification

The agents' orchestration specification is written in the proposed programming language and stored as a text file with the `.api` extension. It serves as a declarative blueprint for how agents exchange messages.

There are no mandatory constructs in the specification, offering adaptability based on the system's requirements. However, a comprehensive specification typically follows a structured order for clarity and maintainability. It begins with holons import, followed by environment configuration, then channels, agents, and finally, one or more execution flows that dictate the MAS dynamic behavior. Below is a sample showcasing the specification syntax:

```
1 // Import holons
2
3 // Configure environment
4
5 // Configure channels
6
7 // Configure agents
8
9 // Configure execution flows
```

Listing 5.10: Agents' orchestration specification structure

Where a concrete example would look as follows:

```
import holon_2

environment .

channel c .

agent a :
    ENV_INPUT -> self
        self -> c

agent b :
        c -> self
        self -> ENV_OUTPUT

start a | b
```

Listing 5.11: Agents' orchestration specification example

The example above describes a system in which `holon_2` is an external holon from which this holon reads outputs through its environment interface. Two agents, `a` and `b`, start in parallel. Whatever is received through this holon's input environment is read by agent `a`. Agent `a` sends messages to a channel named `c`, while agent `b` listens for messages from `c`, processes them, and sends the output to the output environment.

As seen above, to import another holon, include an import statement specifying the holon's name. This represents a link indicating that the current holon should receive messages from the imported holon. This essentially means that if imported holon sends a message to its environment output, the current holon input environment will consume the message. The keyword `import` is used, followed by any valid holon name, indicating that the named holon is available for interaction.

```
import holon_2
```

Listing 5.12: Holon import declaration

The environment in the specification defines how the holon interacts with external systems. It can take one of two formats: forward environment or transformative environment.

A forward environment simply passes data through, meaning that values reaching the environment remain unchanged. No transformation, parsing, or extraction is applied. Its

declaration follows a minimal syntax:

```
environment .
```

Listing 5.13: Transparent environment declaration

A transformative environment, on the other hand, allows for modifications to incoming and outgoing values. It specifies how input data should be extracted or parsed and how output data should be transformed before transmission. The syntax explicitly defines these transformations:

```
environment :
    input  *-> json({'val1':?x})
    output <-* json({'val5':?x})
```

Listing 5.14: Transformative environment declaration

Additionally, the environment can communicate using either `TCP` or `UDP`. The notation for message directionality depends on the chosen protocol:

- UDP communication uses `*->` for sending and `<-*` for receiving messages.
- TCP communication uses `->` for sending and `<-` for receiving messages.

This distinction allows the specification to clearly define how data flows between holons and external systems. A holon may only have one input and output environment, and as a result, name assignment for the environment is not supported.

The channel configuration defines how messages are exchanged within the system. Similar to the environment, channels can operate in a forward or transformative manner. However, unlike the environment, multiple channels can be defined, and each is assigned a unique name following the `channel` keyword.

A forward channel functions as a simple message relay, passing values without modification. Its syntax follows a minimal structure:

```
channel c .
```

Listing 5.15: Transparent channel declaration

A transformative channel, on the other hand, allows for message parsing and transformation. It specifies how input messages are extracted and how they should be formatted before being sent to the next processing stage. The following example demonstrates a channel with transformation:

```
channel c :
    json({"data": ?var}) --> xml(<Abc test="?var" />)
```

Listing 5.16: Transformative channel declaration

In this case, the input message is in JSON format, containing a field named `"data"`, which is mapped to the variable `?var`. The extracted value is then transformed into an XML message, where `?var` is inserted into the test attribute of the `<Abc>` tag. Aside from JSON and XML, environment and channels may also work with REGEX pattern matching. Similar to environments, channels can communicate using either TCP or UDP, with different notation for message directionality.

Agents define active components in the system that send and receive messages through channels, environment, or standard input-output interfaces. Every agent is always associated with a name, which follows the `agent` keyword. Agents can either have a fixed configuration or accept parameters that refer to channels.

A static agent has its communication channels explicitly defined. For example:

```
1 agent a :
2     self -> c
```

Listing 5.17: Agent declaration

In this case, `a` sends messages to the channel `c`. Since `self` appears on the left side of `->`, it means that the agent is the sender, and `c` is the destination channel.

An agent can also take parameters, making it more flexible. When an agent is parameterized, the channels it communicates with are passed as arguments instead of being hardcoded. For example:

```
1 agent a(c) :
2     c -> self
```

Listing 5.18: Parametrized agent declaration

Here, `a` is parameterized with a channel `c`, meaning it listens for messages from `c`. Since `self` is on the right side of `->`, this indicates that the agent is the recipient of messages sent through the channel.

An agent may have multiple communication definitions within its specification, allowing it to both send and receive messages through different channels. For example:

```
1 agent a :
2     c -> self
3     d -> self
4     self -> b
```

Listing 5.19: Agent with multiple communication flows declaration

In this case, `a` listens for messages from both `c` and `d`, while also sending messages to `b`. An agent must always include `self` in its definition, ensuring that it participates in at least one communication flow. If `self` is on the left side of `->`, the agent is sending a message to a channel. If `self` is on the right side, the agent is receiving a message

from a channel. Since agents can only communicate with channels, every message sent or received must be associated with a defined channel.

Execution flows define the sequence in which agents are started within the system. They describe how agents are initialized and managed during execution. There may be multiple execution flows within a specification, allowing for flexible process definitions.

An execution flow always begins with the `start` keyword, followed by the names of the agents to be executed, optionally including operators that define execution behavior.

A previously parameterized agent can be instantiated within an execution flow by providing specific arguments. For example:

```
1 start a(c)
2 start a(d)
```

This means that agent `a` is started twice, once with channel `c` and once with channel `d`, allowing for multiple independent instances.

Agents can also be started in parallel using the | operator:

```
1 start a | b
```

Listing 5.20: Parallel agent startup declaration

In this case, both `a` and `b` are started at the same time, running concurrently.

For sequential execution, where one agent must successfully complete before the next one starts, the `&` operator is used:

```
1 start a & b
```

Listing 5.21: Conditional agent startup on successful completion declaration

Here, `a` is started first. If it completes successfully, then `b` is started.

To define an error-dependent execution, where the second agent runs only if the first one encounters an error, the `!` operator is used:

```
1 start a ! b
```

Listing 5.22: Conditional agent startup on failed completion declaration

This means that `b` will only start if `a` fails.

An agent can also be configured to restart indefinitely once it completes using the `+` operator:

```
1 start a+
```

Listing 5.23: Restartable agent startup declaration

This ensures that `a` will be restarted each time it finishes execution.

5.2 Orchestration platform

The orchestration platform is built using Python technologies, and its primary objective is to consume agents' orchestration specifications through a declarative engine to orchestrate agents effectively. The engine operates recursively when reading the specification, as the specification itself may contain holons, which are components of other specifications.

Within the declarative engine, holons, environments, and channels are core elements specifically designed for message passing and agent connectivity. A holon encapsulates information about the environments, channels, and agents it comprises and is responsible for establishing connections with each element instance using the XMPP protocol and the SPADE framework. XMPP serves as the foundational layer for initiating communication [93, 70, 31].

Once communication is established, the channels, environments, and agents interact using either the UDP or TCP protocol. This hybrid approach enables the system to leverage the strengths of both protocols. Such flexibility allows the orchestration platform to optimize performance based on the specific requirements of the agents' orchestration specification.

Due to the flexibility of the agent architecture in handling various communication contracts, an agent wrapper is introduced. This wrapper functions as a socket, facilitating the conversion of input and output messages between the agent and the platform. It ensures that messages are properly formatted for communication with other entities via TCP or UDP.

The socket is designed to support multiple communication protocols for agent input and output, enabling integration with diverse systems. The supported protocols include: `STDIN`, `FILE` (which translates file-based communication into shell commands), `HTTP`, `WS`, and Netcat.

As a summary, the communication inside the orchestration goes as:

1. XMPP serves as the initial communication layer between holons, channels, environments, and agents. It is primarily responsible for establishing connections, exchanging communication metadata, and tracking state changes across entities.

2. TCP & UDP are the primary transport protocols for passing agent messages (actual content) between agents, channels, and environments.

3. The agent wrapper socket supports various protocols (`STDIN`, `FILE`, HTTP, WS, and Netcat) to facilitate flexible agent communication, allowing integration with different execution environments.

Figure 5.2 illustrates how multiple holons, along with their constituent entities, engage in communication and interaction.

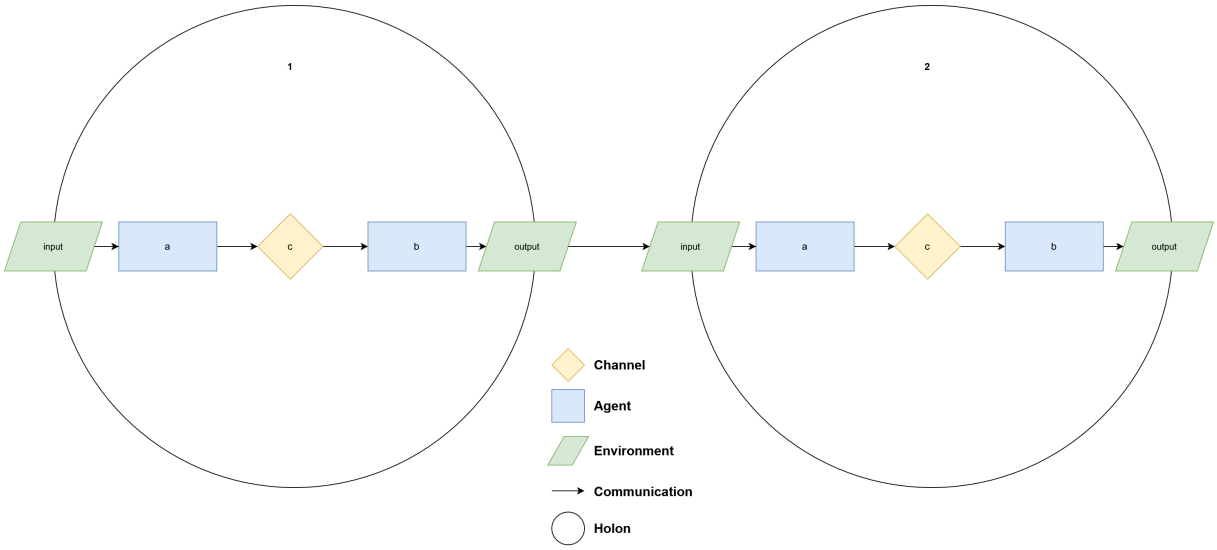

Figure 5.2: `APi` ecosystem

5.2.1 Agents' orchestration specification consumption

To start the orchestration platform and initialize the necessary agents, channels, and environments that constitute a holon, an agents' orchestration specification file must be provided. This file should have an `.api` extension, as it serves as the core configuration for orchestrating interactions within the MAS.

The following function extracts the orchestration specification name from the provided command-line arguments:

```
def extract_orchestration_specification_name() -> str:
    if len(sys.argv) > 2:
        logger.info("Usage: APi [filename.api]")
    else:
        if len(sys.argv) == 2:
            file_name = sys.argv[1]
            splits = file_name.split(".")
            return splits[0]
        else:
            logger.info("Not supported at this time")
            return None
```

Listing 5.24: Agent's orchestration specification loading

The process of programming language grammar generation produces a parser and a lexer, which are used to parse and extract relevant entities from the communication flow specification, as illustrated below:

```
def read_specification_from_file(fl: str) -> APiNamespace:
    stream = FileStream(fl, encoding="utf-8")
    lexer = APiLexer(stream)
    return process(lexer)
```

Listing 5.25: Utilization of lexer and parser to extract values from agents' orchestration specification

A specification may reference other holons, in which case the system recursively loads all associated specifications to capture their interdependencies across multiple files. This approach ensures a comprehensive and interconnected execution of the defined communication flows. The following function facilitates this recursive loading process:

```
def generate_namespaces(configuration_name: str) -> Dict:
    file_name = _ORCHESTRATION_FILE_NAME_TEMPLATE.format(file_name=
        configuration_name)
    file_name = "orchestration_specifications/" + file_name
    return read_specification_files_recursively(file_name)
        _specification_files_recursively(file_name)
```

Listing 5.26: Recursive load of imported holons of agents' orchestration specification

The primary script, `main.py`, is responsible for loading the holon specifications and orchestrating their interactions. It enables effective communication among holons by subscribing them to their respective input and output environments. The process begins by extracting the orchestration specification name, followed by generating namespaces that contain all relevant holon definitions. Each holon is then registered and initialized accordingly:

```
orchestration_specification_name =
    extract_orchestration_specification_name()
if not orchestration_specification_name:
    exit()

ns = generate_namespaces(orchestration_specification_name)
holon_names = list(ns.keys())

rs = APiRegistrationService(orchestration_specification_name)

holons_addressbook = {}
for holon in holon_names:
    h1name, h1password = rs.register(holon)
    holons_addressbook[holon] = {"address": h1name, "password":
        h1password}

for holon, namespace in ns.items():
    agents = namespace.agents
    channels = namespace.channels
    environment = namespace.environment
    execution_plans = namespace.execution_plans
```

```
    holons = namespace.holons
    holon_addresses = {ch: holons_addressbook[ch]["address"] for ch in
        holons}

    creds = holons_addressbook[holon]
    h = APiHolon(
        holon,
        creds["address"],
        creds["password"],
        agents,
        channels,
        environment,
        holon_addresses,
        execution_plans,
    )
    h.start()
```

Listing 5.27: Address book generation for interconnected holons

As previously mentioned, entities within the orchestration platform communicate with one another using the XMPP [93]. This ensures metadata exchange between agents, channels, and environments.

To facilitate this communication, the system includes an `APiRegistrationService` class, which reads the MAS configuration file. This configuration file contains essential details about the system, such as XMPP server settings and port allocations. The name of this configuration file must match the initial communication flow specification file, but with a `.cfg` extension.

Below is an example of a configuration file:

```
registration-services:
  - "rec.foi.hr:49999"
  - "dragon.foi.hr"
port-range:
  min: 3000
  max: 5000
```

Listing 5.28: Example MAS configuration file

The properties defined in this file specify the XMPP server details and define the range of ports that channels and environments can use for establishing TCP and UDP connections. These settings enable efficient communication across different holons within the system.

The `APiRegistrationService` class includes a register method responsible for registering accounts on the XMPP server for each individual entity. This ensures that every

holon has unique credentials for authentication and communication. The registration process dynamically generates a username and password for each holon:

```
    def register(self, holon_name: str) -> str:
        username = "%s_%s_%s" % (self.mas_name, holon_name, str(uuid4().
            hex))
        password = str(uuid4().hex)
        host = settings.xmpp_host

        return ("%s@%s" % (username, host), password)
```

Listing 5.29: Authentication details generation for XMPP communication

5.2.2 Holon

Holon primarily relies on XMPP and acts as a top-level coordinator for all agents, channels, and an optional environment in this MAS. By centralizing control in holon, the system ensures that agents are not burdened with discovering each other's addresses or orchestrating execution logic. Instead, each agent simply carries out its individual tasks once it receives the necessary parameters and addresses from holon [93, 70, 31].

The class `APiHolon` inherits from a base class called `APiCommunication`, which already handles lower level message sending and receiving. The constructor of holon begins by setting up references to environments, channels, and agent instances. Every element involved in communication, regardless of type, requires a unique address and password in the XMPP domain. The class `APiRegistrationService` handles these registrations.

Below is a relevant portion of holon's constructor showing the initial setup:

```
class APiHolon(APiCommunication):

    def __init__(
        self,
        holonname,
        name,
        password,
        agents,
        channels,
        environment,
        holons_addressbook,
        execution_plans,
    ):
        self.token = str(uuid4().hex)
        super().__init__(name, password, str(uuid4().hex))
        self.holonname = holonname
        self.address = str(self.jid.bare())
```

```
18          self.namespace = APiNamespace()
19          self.registrar = APiRegistrationService(holonname)
20
21          self.environment = None
22          if environment:
23              self.setup_environment(environment)
24
25          self.channels = {}
26          for c in channels:
27              self.setup_channel(c)
28
29          self.agent_types = {}
30          self.agents = {}
31          for a in agents:
32              self.create_agent_types_map(a)
33
34          self.holons = holons_addressbook
35
36          self.execution_plans = None
37          if len(execution_plans) > 0:
38              self.setup_execution(execution_plans)
39
40          self.all_channels_listening = False
41          self.all_agents_listening = False
42
43          self.all_started = False  # Indicate if execution plan has been
                started already
44          self.start_env_and_channels()
```

Listing 5.30: `APiHolon` constructor

When the holon invokes methods like `setup_channel` or `setup_agent`, it requests credentials from the `registrar` and then constructs a command line that launches the corresponding script as a separate process. Below is the channel initialization code:

```
1  def setup_channel(self, channel: Dict) -> None:
2      address, password = self.registrar.register(channel["name"])
3      logger.debug(f"Registering channel {channel['name']}")
4
5      # NOTE: This should be updated if channel.py is moved around
6      channel["cmd"] = (
7          'poetry run python ../src/agents/channel.py "%s" "%s" "%s" "%s"
               "%s" "%s" "%s" "%s" "%s"'
8          % (
9              channel["name"],
10             address,
11             password,
12             self.address,
```

```
13              self.token,
14              json.dumps((self.registrar.min_port, self.registrar.max_port
                    )),
15              channel["protocol"],
16              json.dumps(channel["input"]).replace('"', '""'),
17              json.dumps(channel["output"]).replace('"', '""'),
18          )
19      )
20      channel["address"] = address
21      channel["status"] = "setup"
22      self.channels[channel["name"]] = channel
```

Listing 5.31: Channel initialization

The environment is similarly set up via `setup_environment`, but it is typically used to simulate or interact with external conditions. Once holon has registered each channel and environment, it spawns them in threads using `start_env_and_channels`:

```
1  def start_env_and_channels(self) -> None:
2      if self.environment:
3          cmd = shlex.split(self.environment["cmd"])
4          logger.debug(f"Running environment: {self.environment.get('name
               ')}")
5          self.environment["instance"] = Thread(target=self.
               start_basic_agent_thread, args=(cmd,))
6          self.environment["instance"].start()
7          self.environment["status"] = "started"
8
9      for c in self.channels.values():
10         cmd = shlex.split(c["cmd"])
11         logger.debug(f"Running channel: {c.get('name')}")
12         c["instance"] = Thread(target=self.start_basic_agent_thread,
               args=(cmd,))
13         c["instance"].start()
14         c["status"] = "started"
```

Listing 5.32: Environments and channels startup

Each thread eventually calls `subprocess.Popen(...)` inside `start_basic_agent_thread`, allowing channels or environments to run in separate processes. They subsequently send messages to holon indicating that they are ready or listening.

For agents, holon maintains a mapping in `self.agent_types`, and each agent is registered and prepared using a command string similar to that of a channel. In `setup_agent`, holon retrieves or generates a unique address for the agent, possibly adjusts that agent's communication flows with `adjust_flows_by_args`, and then constructs a startup command referencing `agent.py`:

```
def setup_agent(
    self, agent_type: str, id: str = None, plan_id: str = None, params:
        List = None
) -> None:
    if not id:
        id = uuid4().hex

    agent = deepcopy(self.agent_types[agent_type])
    logger.debug(f"Registering agent {agent['name']}")
    address, password = self.registrar.register(agent["name"])
    if params:
        flows = self.adjust_flows_by_args(agent["args"], params, agent["
            flows"])
    else:
        flows = agent["flows"]

    # NOTE: This should be updated if agent.py is moved around
    agent["cmd"] = (
        'poetry run python ../src/agents/agent.py "%s" "%s" "%s" "%s" "%
            s" "%s" "%s"'
        % (
            agent["name"],
            address,
            password,
            self.address,
            self.holonname,
            self.token,
            json.dumps(flows).replace('"', '\\"'),
        )
    )
    agent["address"] = address
    agent["status"] = "setup"
    agent["id"] = id
    agent["plan_id"] = plan_id
    self.agents[id] = agent
```

Listing 5.33: Agent initialization

The address book is a structure in holon that stores references to channels, environments, agents, and other holons by an identifier. Once an agent finishes its initialization and is ready, it explicitly queries holon over XMPP for the specific addresses it needs. Below is a snippet from the `RequestForAddress` behavior illustrating this flow:

```
class RequestForAddress(CyclicBehaviour):

    async def run(self) -> None:
        msg = await self.receive(timeout=0.1)
        if msg:
```

```
6             if self.agent.verify(msg):
7             logger.debug("(RequestForAddress) Message verified,
                 processing ...")
8                 channel = msg.metadata["channel"]
9                 metadata = deepcopy(self.agent.query_message_template)
10                metadata["in-reply-to"] = msg.metadata["reply-with"]
11                metadata["agent"] = channel
12
13                try:
14                    if (
15                        channel == "ENVIRONMENT" or channel == self.
                             agent.holonname
16                    ) and self.agent.environment is not None:
17                        address = self.agent.environment["address"]
18                    else:
19                        address = self.agent.channels[channel]["address"
                             ]
20
21                    logger.debug(f"Found channel {channel} address is {
                         address}")
22
23                    metadata["success"] = "true"
24                    metadata["address"] = address
25                except KeyError:
26                    logger.debug(f"Channel {channel} not found")
27                    metadata["success"] = "false"
28                    metadata["address"] = "null"
29                await self.agent.schedule_message(str(msg.sender),
                     metadata=metadata)
```

Listing 5.34: Holon SPADE behaviour used for agent requesting address of other entity

A distinguishing feature is how holon interprets a custom execution flow specification that defines the order and conditions under which agents execute. Holon calls `resolve_execution_plan` to parse the execution flow. Below is an excerpt:

```
1 def resolve_execution_plan(execution_plan: str) -> Tuple[str, Dict, List
     ]:
2     # find out paralel flows
3     parallel = execution_plan.split("|")
4
5     # trim and remove unused chars
6     cleaned_exp = [_extract_args_and_clean_up(exp) for exp in parallel]
7
8     # resolve agents and operations
9     parallel_flows = [_get_agents_and_operations(item) for item in
         cleaned_exp]
```

```
10
11      # get initial agents to run
12      initial_agents = _get_initial_agents_to_run(parallel_flows)
13
14      # flattening all agents across different parallel flows into a high-
            level map (no nesting)
15      agents = {}
16      for flow in parallel_flows:
17          agents = {**agents, **flow}
18
19      return {
20          "id": uuid4().hex,
21          "plan": agents,
22          "initial_agents_to_run": initial_agents,
23          "started": False,
24      }
```

Listing 5.35: Parsing execution flow

Helper functions (`_extract_args_and_clean_up` and `_get_agents_and_operations`) ensure that each agent in the plan is assigned:

- A unique ID
- An optional operator indicating the condition for triggering the next agent
- A potential succeeding agent ID, linking to another agent that should launch next

Once the plan is parsed, holon knows which agents need to start up first. It launches them using `start_initial_agents`, while any subsequent agents start only after holon detects that the preceding agents have finished (with or without error), or in parallel if that is indicated in the execution flow.

When agents finish, the `agent_finished` method in holon processes the finish event to decide the next step:

```
1  def agent_finished(self, a_id: str, plan_id: str, status_code: int) ->
       None:
2      plan = None
3      for p in self.execution_plans:
4          if p["id"] == plan_id:
5              plan = p
6      agent_exec = plan["plan"].get(a_id, None)
7
8      # may not be needed, unless errored
9      self.agents[a_id]["status"] = "stopped"
10
11     operator = agent_exec["operator"]
```

```
12      succeeding_agent_id = None
13      # start agent again
14      if operator == "+":
15          succeeding_agent_id = a_id
16      # start new agent if current errors
17      elif operator == "!":
18          if status_code != 0:
19              succeeding_agent_id = agent_exec["succeeding_agent_id"]
20      # start new agent if current succeeds
21      elif operator == "&":
22          if status_code == 0:
23              succeeding_agent_id = agent_exec["succeeding_agent_id"]
24      # no matter the status, start the new agent
25      elif not operator:
26          succeeding_agent_id = agent_exec["succeeding_agent_id"]
27
28      succeeding_agent = None
29      if succeeding_agent_id is not None:
30          for a in self.agents.values():
31              if a["id"] == succeeding_agent_id:
32                  succeeding_agent = a
33
34      if succeeding_agent is not None:
35          self.run_agent_thread(succeeding_agent, "dependant")
```

Listing 5.36: Agent on finished handler

Holon logic utilizes SPADE behaviors to stay informed about the agents' life cycle. For instance, the `FinishedAgents` behavior is responsible for detecting messages with the state `finished`, verifying them to ensure authenticity, and updating statuses to `stopped`. If an error is reported, holon logs it. Likewise, other behaviors such as `GetReadyAgents` and `GetListeningAgents` handle messages from agents, channels, environments indicating readiness. Below is the implementation of the `FinishedAgents` behavior:

```
1  class FinishedAgents(CyclicBehaviour):
2      async def run(self) -> None:
3          msg = await self.receive(timeout=0.1)
4          if msg:
5              if self.agent.verify(msg):
6                  logger.debug("(FinishedAgents) Message verified,
                       processing ...")
7                  agent = self.agent.agent_name_from_address(msg.sender.
                       bare())
8
9                  if msg.metadata["error-message"] != "null":
10                     logger.debug(
11                         f"Agent {agent} finished with error {str(msg['
```

```
                        error-message'])}"
12                  )
13              else:
14                  logger.debug(f"Agent {agent} finished gracefully.")
15              self.agent.agents[agent]["status"] = "stopped"
16
17              # sending message to ack that agent has stopped, so they
                    can terminate
18              metadata = deepcopy(self.agent.confirm_message_template)
19              metadata["action"] = "finish"
20              metadata["in-reply-to"] = msg.metadata["reply-with"]
21
22              await self.agent.schedule_message(str(msg.sender),
                   metadata=metadata)
```

Listing 5.37: Holon SPADE behavior used for handling agent completion

This decentralized messaging allows each agent to focus solely on its domain tasks. Agents report their status (start, error, or completion) back to holon via XMPP, and holon coordinates subsequent steps according to the execution plan. This architecture provides a clear separation of concerns, with holon orchestrating the work:

- Registration: Creating unique addresses and credentials for each entity
- Process spawning: Launching channels, environments, and agents as separate processes
- Address book distribution: Selectively providing element instances with only the addresses they actually need
- Execution flow: Specifying the order in which agents start and how they depend on one another
- Life cycle event processing: Listening for agent statuses (ready, error, finished) and reacting to them dynamically

Agents do not need to discover one another's addresses or interpret complex orchestration logic. After registering themselves and declaring they are ready, they carry out their specialized tasks and then inform holon of any relevant results, including errors. Any restarts, fallback logic, or parallel coordination is handled by holon, which remains the global authority on how the entire MAS proceeds. This approach keeps the system highly modular, making it simple to add new agents, introduce alternate flows, or reconfigure channels and environments without modifying the agents' internal code.

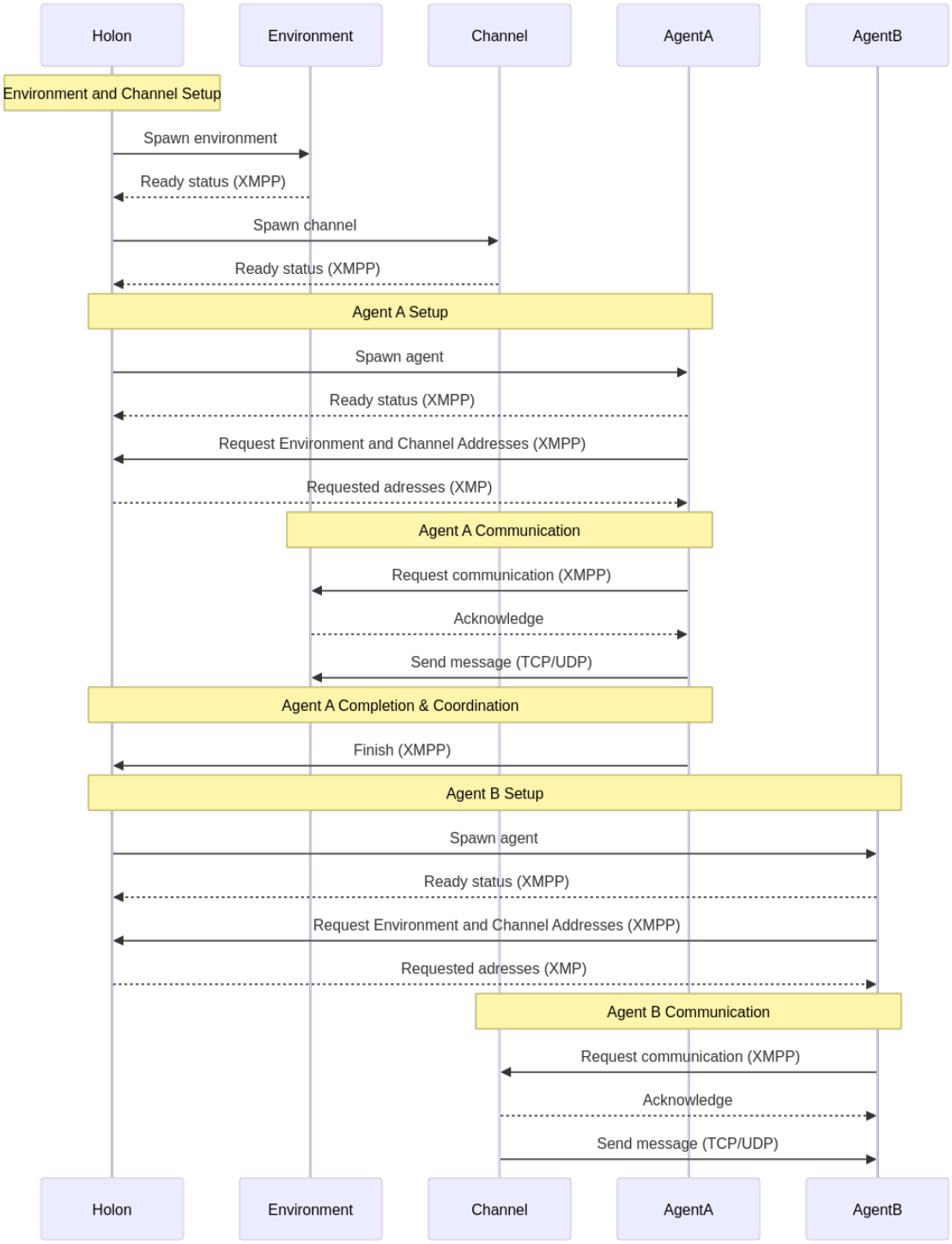

Figure 5.3: Communication between different entities in a holon

Figure 5.3 illustrates the communication flow within holon among environment, channel, and agents. Environment and channel are initialized first, ensuring they are active before any agents come online. Once those are running, holon sets up agent `A`, which notifies holon that it is ready and then queries holon over XMPP for the channel and environment addresses. Agent `A` uses these addresses to exchange messages via TCP or

UDP as needed. Once agent `A` completes its task, it reports its status back to holon, prompting holon to determine whether additional agents should be spawned. In this example, agent `B` follows the same sequence: requesting addresses from holon, engaging in communication, and finally reporting completion via XMPP. If holon decides to halt the system, it can stop all running entities by sending them a stop signal, ensuring a coordinated shutdown.

5.2.3 Channel

The channel serves as a communication proxy between agents. It facilitates interaction using either UDP or TCP protocols, ensuring message exchange. Additionally, it can extract values from incoming messages and map them to the expected output, allowing for flexible data transformation.

The `APiChannel` class implements the described communication mechanisms. It is built on the SPADE framework, utilizing structured behaviors to handle agent interactions. The `APiChannel` instance enables agents to subscribe for incoming messages or attach as a sender, ensuring that messages are efficiently routed and optionally transformed [70].

The `APiChannel` class extends `APiBaseChannel`, which provides the foundational implementation for managing communication channels, defining mapping strategies, and handling server-client interactions. `APiBaseChannel` is responsible for initializing the communication infrastructure, setting up transformation mechanisms, and managing socket connections, while `APiChannel` builds upon this by adding specific behaviors for handling subscription and attachment of agents.

The `APiChannel` implements SPADE behaviors to manage the initial setup of agent interactions. The two primary behaviors are:

- `SubscriptionRequest`: Allows an agent to request permission to listen for messages sent to the channel, as well as to send messages.
- `AgentMessageListening`: A behavior responsible for listening for incoming messages and forwarding them to subscribed agents.

Each behavior ensures structured interaction through XMPP, allowing agents to communicate in a standardized manner. When an agent sends a message with the `subscribe` performative to the channel via the XMPP, it indicates that the agent wishes to listen for incoming messages. The agent's connection is then registered, allowing it to receive messages forwarded by the channel. On the other hand, when an agent sends a message with the `request` performative, it signals its intent to send messages to the channel. The channel will then handle the transmission of these messages to any subscribed agents over the selected protocol [93].

The channel is capable of transforming incoming messages before forwarding them. Supported transformation mechanisms include:

- REGEX: Extracts values based on patterns.
- JSON: Extracts values from JSON.
- XML: Extracts values from XML.

This mechanism allows messages to be converted into the required format before being forwarded to the appropriate subscribers.

```
if self.input.startswith("regex("):
    reg = self.input[6:-1]
    self.input_re = re.compile(reg)
    self.map = self.map_re
elif self.input.startswith("json("):
    self.input_json = self.input[5:-1]
    self.kb.query("use_module(library(http/json))")
    cp = self.input_json
    replaces = {}
    for var in self.var_re.findall(self.input_json):
        rpl = self.REPL_STR % var
        replaces[rpl[1:-1]] = var
        cp = cp.replace(var, rpl)
    query = " APIRES = ok, open_string( '%s', S ), json_read_dict( S, X
        ). " % cp
    res = self.kb.query(query)
    prolog_json = res[0]["X"]
    for k, v in replaces.items():
        prolog_json = prolog_json.replace(k, "X" + v[1:])

    self.input_json = prolog_json

    self.map = self.map_json
elif self.input.startswith("xml("):
    self.input_xml = self.input[4:-1]
    cp = self.input_xml
    replaces = {}
    for var in self.var_re.findall(self.input_xml):
        rpl = self.REPL_STR % var
        replaces[rpl[1:-1]] = var
        cp = cp.replace(var, rpl)

    for k, v in replaces.items():
        input_xml = cp.replace(k, "X" + v[1:])

    input_xml = xmltodict.parse(input_xml)
```

```
36      self.input_xml = str(input_xml).replace(" ", "").replace("'", "").
            replace("@", "")
37
38      self.map = self.map_xml
```

Listing 5.38: Input to output transformation mechanism

The channel dynamically manages network sockets, determining available ports and assigning them based on the required protocol. The following method ensures that a free port is identified before initiating a server:

```
1  def get_free_port(self, protocol: str) -> int:
2      if protocol == "tcp":
3          sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4      else:
5          sock = socket.socket(type=socket.SOCK_DGRAM)
6      port = self.min_port
7      while port <= self.max_port:
8          try:
9              sock.bind(("", port))
10             sock.close()
11             return port
12         except OSError:
13             port += 1
14     raise IOError("No free ports in range %d - %d" % (self.min_port,
           self.max_port))
```

Listing 5.39: Dynamic port allocation

A new UDP or TCP server is instantiated based on the specified protocol. Agents that subscribe or attach are stored in internal data structures for efficient message forwarding. When an agent attaches and sends messages to the channel, the `AgentMessageListening` behavior ensures the message reaches all subscribed agents.

```
1  class AgentMessageListening(CyclicBehaviour):
2
3      async def run(self) -> None:
4          def iter_clients(srv):
5              if self.agent.protocol == "udp":
6                  yield srv
7              else:
8                  try:
9                      c, a = srv.sock.accept()
10                     is_udp = True if self.agent.protocol == "udp" else
                           False
11                     client = nclib.Netcat(sock=c, server=a, udp=is_udp)
12                     yield client
13                 for client in srv:
```

```
14                          yield client
15                  except Exception as e:
16                      if str(e) != "timed out":
17                          logger.error(f"Error accepting client: {e}")
18                      return
19
20          if self.agent.write_servers:
21              for srv in self.agent.write_servers:
22                  srv.sock.settimeout(0.1)
23                  for client in iter_clients(srv):
24                      # TODO should put in a method instead
25                      if self.agent.protocol == "udp":
26                          result = None
27                          try:
28                              result, _ = client.sock.recvfrom(1024)
29                          except Exception as e:
30                              logger.error(f"Error receiving from client:
                                    {e}")
31                              pass
32                      else:
33                          result = client.recv_until(self.agent.delimiter,
                                timeout=0.1)
34                      logger.info(f"Received result: {result}")
35                      if result:
36                          logger.info(f"Mapping result: {result}")
37                          msg = self.agent.map(result.decode())
38                          print("msasdg", result.decode())
39                          logger.info(f"Sending msg: {msg}")
40
41                          self.agent.send_to_subscribed_read_agents(msg.
                                encode())
```

Listing 5.40: Channel SPADE behavior used for incoming messages handling

5.2.4 Environment

Environment is similar to a channel in the sense that it proxies messages between agents and other holon environments. Each holon can have one input and one output environment. Other entities can either listen to messages or send messages to both input and output environments. While the channel mainly facilitates direct communication between agents, the environment plays a broader role by structuring the input and output message flow within a holon-based system [31].

The `APiEnvironment` agent acts as a communication hub between agents and a holon, managing both input and output message flows. It extends the `APiBaseChannel` and

provides structured communication using different protocols. Unlike the channel, which primarily facilitates direct message exchange between agents, the environment plays a broader role by handling structured input and output streams for a MAS.

The environment supports both TCP and UDP protocols for incoming and outgoing messages, dynamically setting up dedicated servers for each communication type. Agents can either subscribe to listen to messages sent through the environment or attach themselves to send messages.

The environment dynamically provisions network sockets for its subscribers and attached agents. The following method is responsible for managing agent communication by retrieving a dedicated subscription server based on whether it is handling input or output:

```
def get_read_server(self, env_type: str, protocol: str) -> tuple[str,
    str, int, str]:
     instance = (
         self.input_subscribe_socket_server
         if env_type == "input"
         else self.output_subscribe_socket_server
     )

     srv = instance["server"]
     ip = instance["ip"]
     port = instance["port"]
     protocol = instance["protocol"]

     return srv, ip, port, protocol
```

Listing 5.41: Retrieving a network socket given incoming request

The environment also distinguishes between agents that only listen and those that send data. The following function ensures messages are correctly forwarded to all relevant subscribers based on whether they are listening for input or output:

```
def send_to_subscribed_read_agents_and_holons(self, env_type: str, msg:
    bytes) -> None:
     socket_clients = (
         self.socket_clients["input_subscribe_socket_clients"]
         if env_type == "input"
         else self.socket_clients["output_subscribe_socket_clients"]
     )
     s_server = (
         self.input_subscribe_socket_server
         if env_type == "input"
         else self.output_subscribe_socket_server
     )
```

```
12      protocol = self.input_protocol if env_type == "input" else self.
            output_protocol
13
14      if protocol == "udp":
15          for client in socket_clients:
16              s_server["server"].respond(msg, client)
17      else:
18          closed_clients = []
19          for idx, client in enumerate(socket_clients):
20              try:
21                  client.sendline(msg)
22              except Exception as ex:
23                  logger.info("Run into error sending a msg over socket",
                        ex)
24                  closed_clients.append(idx)
```

Listing 5.42: Forwarding messages to subscribers

Additionally, the `APiEnvironment` class instance features SPADE behaviors, including a `SubscriptionRequest` behavior for managing agent subscriptions and a `HolonMessageListening` behavior for relaying received messages to the appropriate recipients. These behaviors ensure that message flow is handled asynchronously and efficiently [70].

The `SubscriptionRequest` behavior is responsible for handling agent or holon requests to listen to messages or send messages to the environment. When an agent or a holon sends a subscription request, the behavior verifies the request and assigns the appropriate input or output subscription. It determines whether the agent wants to listen to incoming messages or transmit messages into the environment. If the verification succeeds, the environment responds with the relevant communication parameters, such as IP address, port, and protocol type, ensuring the agent is properly registered for interaction. This mechanism provides secure and structured access control over message exchange within the environment.

The following code snippet demonstrates how the `SubscriptionRequest` behavior processes agent requests:

```
1   class SubscriptionRequest(CyclicBehaviour):
2
3       async def run(self):
4           msg = await self.receive(timeout=0.1)
5           if msg:
6               if self.agent.verify(msg):
7                   logger.debug("(Subscribe) Message verified, processing
                        ...")
8                   metadata = deepcopy(self.agent.agree_message_template)
9                   metadata["in-reply-to"] = msg.metadata["reply-with"]
10                  # subscribing to environment input
```

```
11                  if msg.metadata["performative"] == "subscribe_to_input":
12                      metadata["agent"] = "ENV_INPUT"
13                      metadata["type"] = "input"
14                      _, ip, port, protocol = self.agent.get_read_server(
15                          "input", self.agent.input_protocol
16                      )
17                      logger.info("ADDED input subscribe server", ip, port
                            )
18                  # subscribing to environment output
19                  elif msg.metadata["performative"] == "
                        subscribe_to_output":
20                      metadata["agent"] = "ENV_OUTPUT"
21                      metadata["type"] = "input"
22                      _, ip, port, protocol = self.agent.get_read_server(
23                          "output", self.agent.output_protocol
24                      )
25                      logger.info("ADDED output subscribe server", ip,
                            port)
26                  # attaching to environment output
27                  elif msg.metadata["performative"] == "request_to_input":
28                      metadata["agent"] = "ENV_INPUT"
29                      metadata["type"] = "output"
30                      _, ip, port, protocol = self.agent.get_write_server(
31                          "input", self.agent.input_protocol
32                      )
33                      logger.info("ADDED input attach server", ip, port)
34                  elif msg.metadata["performative"] == "request_to_output"
                        :
35                      metadata["agent"] = "ENV_OUTPUT"
36                      metadata["type"] = "output"
37                      _, ip, port, protocol = self.agent.get_write_server(
38                          "output", self.agent.output_protocol
39                      )
40                      logger.info("ADDED output attach server", ip, port)
41                  else:
42                      logger.debug("Unknown message")
43                      metadata = self.agent.refuse_message_template
44                      metadata["in-reply-to"] = msg.metadata["reply-with"]
45                      metadata["reason"] = "unknown-message"
46                      await self.agent.schedule_message(str(msg.sender),
                            metadata=metadata)
47
48                  if msg.metadata.get("external", "") == "True":
49                      metadata["agent"] = self.agent.holon_name
50
51                  metadata["server"] = ip
52                  metadata["port"] = port
```

```
53                 metadata["protocol"] = protocol
54                 await self.agent.schedule_message(str(msg.sender),
                       metadata=metadata)
55                 await asyncio.sleep(0.1)
```

Listing 5.43: Channel SPADE behavior used for incoming messages handling

5.2.5 Agent wrapper

The agent wrapper is a specialized process responsible for bridging the communication gap between an actual agent and the orchestration platform. It exchanges messages with holon over XMPP to receive or report status changes and to request the addresses of other entities (channels and environments). Simultaneously, it uses TCP and UDP to transmit messages between environments and channels, while enabling communication with the actual agent over various protocols, including Netcat, WS, HTTP, `STDIN`, and the filesystem. Agent communication defines input and output separately, allowing flexibility when the input and output protocols differ [93].

This concept of a wrapper arises when the user already has a previously developed agent that runs through UNIX [29], Docker [110], or Kubernetes [18]. The wrapper stands in front of this existing software i.e. agent, capturing inputs from the MAS and forwarding them to the agent, while also capturing the agent's outputs and sending them back out to other participants as needed. Below is a code snippet (from the `RequestAdresses` behavior) illustrating how the wrapper queries holon for the specific addresses of channels or environments:

```
1  class RequestAdresses(OneShotBehaviour):
2
3      async def run(self) -> None:
4          # waiting for address book containing input channels from holon
5          logger.debug(f"Inputs: {self.agent.input_channel_query_buffer}")
6          for inp in self.agent.input_channel_query_buffer:
7              metadata = self.agent.query_msg_template
8              metadata["reply-with"] = str(uuid4().hex)
9              metadata["channel"] = inp
10             await self.agent.schedule_message(self.agent.holon, metadata
                   =metadata)
11
12         logger.debug(f"Outputs: {self.agent.output_channel_query_buffer}
               ")
13         for out in self.agent.output_channel_query_buffer:
14             logger.debug(f"Looking up channel {out} in addressbook")
15             # in case we retrieved the channel from input channels
                   address book batch
```

```
16              try:
17                  channel = self.agent.address_book[out]
18                  logger.debug(f"Got channel {out} address {channel}")
19                  await asyncio.sleep(0.1)
20              except KeyError:
21                  logger.debug(f"Could not find channel {out} in address
                        book, querying")
22                  metadata = self.agent.query_msg_template
23                  metadata["reply-with"] = str(uuid4().hex)
24                  metadata["channel"] = out
25                  await self.agent.schedule_message(self.agent.holon,
                        metadata=metadata)
26
27          if (
28              len(self.agent.input_env_query_buffer) > 0
29              or len(self.agent.output_env_query_buffer) > 0
30          ):
31              metadata = self.agent.query_msg_template
32              metadata["reply-with"] = str(uuid4().hex)
33              metadata["channel"] = "ENVIRONMENT"
34              await self.agent.schedule_message(self.agent.holon, metadata
                    =metadata)
```

Listing 5.44: Agent SPADE behavior used for requesting addresses from holon

`inp` references a channel name or environment that was specified in the agent's communication flow definitions. When holon replies, the agent requests a connection with the channel or environment through XMPP. The channel then establishes either a TCP or UDP connection, depending on what was requested. These connections allow the wrapper to facilitate communication between the agent and the MAS. The same pattern applies when it needs the environment's address, allowing the agent wrapper to treat environment interactions just like a specialized channel.

Once the wrapper establishes connections, it distinguishes between channels where it sends messages and those where it subscribes to receive messages. For channels where the agent sends messages, it applies an attach behavior, ensuring data is forwarded correctly. On the other hand, for channels where the agent listens for incoming messages, it subscribes to them to process received data. A snippet from the code shows how the wrapper classifies channels in the `subscribe_to_channel` method:

```
1  def subscribe_to_channel(self, channel: str, channel_type: str) -> None
       :
2      # TODO: Implement channel subscription (sender, receiver)
3      # Channel types:
4      # STDIN -> reads input from stdin
5      # STDOUT/STDERR -> writes output to STDIN/STDERR
```

```
    # <name> -> gets instructions from channel on how
    #           to connect (via Netcat)
    if channel_type == "input":
        if channel == "STDOUT":
            err = "Input cannot be STDOUT"
            raise APiChannelDefinitionError(err)
        elif channel == "STDERR":
            err = "Input cannot be STDERR"
            raise APiChannelDefinitionError(err)
        elif channel == "STDIN":
            self.start_shell_client(prompt=True, await_stdin=True)
        elif channel == "ENV_INPUT":
            self.input_env_query_buffer.append(channel)
        else:
            self.input_channel_query_buffer.append(channel)
    elif channel_type == "output":
        if channel == "STDOUT":
            self.start_shell_client(print_stdout=True)
        elif channel == "STDERR":
            self.start_shell_client(print_stderr=False)
        elif channel == "STDIN":
            err = "Output cannot be STDIN"
            raise APiChannelDefinitionError(err)
        elif channel == "ENV_OUTPUT":
            self.output_env_query_buffer.append(channel)
        else:
            self.output_channel_query_buffer.append(channel)
```

Listing 5.45: Subscribe to channel

The agent can initiate message sending based on its defined logic either proactively or in response to received inputs. It can also wait for messages from the environment, another channel, or be controlled via `STDIN`, where a user or script feeds commands directly. This flexibility arises because the agent wrapper sets up multiple avenues for data flow. On the output side, for instance, when the agent produces data as part of its execution, it forwards the output to a configured channel. Below is a excerpt illustrating how output is handled when the agent sends data to a channel:

```
async def output_callback(self, data: bytes) -> None:
    self.shell_buffer.append(data)
    logger.info(f"About to send {data.encode()} to attached channels")
    for channel, srv in self.output_channel_servers.items():
        logger.info(f"Sending {data.encode()} to {channel}...")
        is_udp = srv["protocol"] == "udp"
        sent = False
        srv["socket"] = nclib.Netcat((srv["server"], srv["port"]), udp=
            is_udp)
```

```
9           while not sent:
10              try:
11                  srv["socket"].sendline(data.encode())
12                  sent = True
13                  logger.info(f"Done sending to channel {channel}")
14              except (BrokenPipeError, ConnectionResetError):
15                  logger.error(f"Error sending {data} (BROKEN PIPE)")
16                  logger.info("Attempting to reconnect")
17                  srv["socket"] = nclib.Netcat((srv["server"], srv["port"
                        ]), udp=is_udp)
18      if data == self.output_delimiter:  # TODO: Verify this
19          self.service_quit("End of output")
```

Listing 5.46: Agent sending messages to attached channels

In this snippet, `output_callback` is called whenever the agent generates data for external transmission. The wrapper receives this data and iterates over all configured output channels. Each channel is associated with a TCP or UDP socket, and the wrapper sends the outgoing message. This ensures reliable transmission of the agent's outputs. When the agent receives incoming messages from the environment, a channel, or `STDIN`, the wrapper directs them into the agent's input logic for processing.

The agent wrapper's primary responsibility is to launch the actual agent process and bridge all communication contracts between that agent and the rest of the MAS. It inspects an agent descriptor file (with `.ad` extension) to determine how to start the underlying agent. Agents must be started by the orchestration platform, and there is no support for interacting with agents that are already running. The excerpt below shows part of the `_load` method, where the wrapper reads the descriptor and constructs the startup command accordingly:

```
1  def _load(self, fh: io.TextIOWrapper) -> None:
2      ...
3      if self.type == "unix":
4          self.cmd = self.descriptor["agent"]["start"]
5      elif self.type == "docker":
6          name = self.descriptor["agent"]["name"]
7          cmd = self.descriptor["agent"]["start"]
8
9          self.cmd = f"docker run -a stdin -a stdout -i -t {name} {cmd}"
10     elif self.type == "kubernetes":
11         name = self.descriptor["agent"]["name"]
12         cmd = self.descriptor["agent"]["start"]
13
14         self.cmd = (
15             f"kubectl run {name} --rm -it --restart=Never --image={name}
                   --command -- {cmd}"
```

```
16          )
17      ...
```

Listing 5.47: Load agent definition and start an agent

When the descriptor indicates "UNIX", the wrapper's `self.cmd` just becomes whatever local command the user has specified in the `.ad` file. If it says "docker", the wrapper constructs a Docker command, binding standard input and output so it can capture the container's streams. For "kubernetes", it similarly forms a `kubectl` run command [110, 18, 29].

Once the agent is launched, the wrapper processes various input and output formats, including HTTP, WSs, Netcat, `STDIN`, and local file. For each combination of input and output, there is a specialized function (or set of functions) that ensures data conversion appropriately. For instance, if an agent wrapper is configured to write data to an HTTP endpoint of an agent but expects a response through a Netcat socket, the wrapper references the corresponding entries in `process_descriptor` to manage the communication flow:

```
1  elif self.input_type[:4] == "HTTP" and self.output_type[:6] == "NETCAT":
2      url = http_re.findall(self.input_type)[0]
3      self.http_url = url
4      host, port, udp = netcat_re.findall(self.output_type)[0]
5      self.nc_host = host
6      self.nc_port = int(port)
7      self.nc_udp = udp != ""
8      self.input = self.input_http
9
10     self.httpnc_thread = Thread(target=asyncio.run, args=(self.
           input_httpnc_run(self.cmd),))
11     self.httpncrec_thread = Thread(
12         target=self.read_nc, args=(self.nc_host, self.nc_port, self.
               nc_udp)
13     )
```

Listing 5.48: Configuring agent wrapper input and output communication

In this snippet, the wrapper checks if the agent is configured to receive inputs via HTTP and send outputs through a Netcat socket. It extracts the HTTP URL from `self.input_type` and parses the Netcat host, port, and UDP flag from `self.output_type`. The wrapper then sets up `self.input_http` as the input handler and initializes two threads: one (`self.httpnc_thread`) to process incoming HTTP data asynchronously and another (`self.httpncrec_thread`) to handle reading responses from the Netcat socket. This setup ensures that data received via HTTP is processed and that responses are transmitted back through Netcat.

By maintaining this input-to-output mapping in a single place, the agent wrapper ensures that developers can plug in an existing command-line or containerized application, specify how it should receive inputs (HTTP, file, `STDIN`, etc.) and where it should send its outputs (WS, Netcat, etc.), and have the wrapper seamlessly establish and route those connections. This makes it possible to integrate a wide variety of software within the MAS, whether it is a simple UNIX process, a Docker container, or a Kubernetes pod, without modifying the underlying application code to communicate over HTTP, WSs, or other protocol [110, 18, 29].

Once the agent wrapper finishes relaying data or encounters an issue, it notifies holon of the outcome. The agent descriptor file (`.ad`) may include conditions for detecting completion: for instance, a certain end delimiter in the agent's output, or a specific line of text signaling termination. The wrapper continuously monitors these conditions and, when one is met, it executes its shutdown sequence. Below is an excerpt of code in which the wrapper handles completion and sends a finished event to holon:

```
def service_quit(self, msg=""):
    self.say(msg)
    self.input_ended = True

async def service_quit_run(self) -> None:
    while not self.input_ended:
        await asyncio.sleep(0.1)

    ...

    metadata = deepcopy(self.inform_msg_template)
    metadata["reply-with"] = str(uuid4().hex)
    metadata["status"] = "finished"
    metadata["error-message"] = "null"
    await self.schedule_message(self.holon, metadata=metadata)
```

Listing 5.49: Agent wrapper signaling completion of its execution

In this snippet, `service_quit` sets a flag indicating that no more data is expected, allowing the wrapper to terminate active threads. The `service_quit_run` coroutine then gathers final status information and sends a finishing message to holon. If an error occurred (for example, a process crash or broken socket), the wrapper replaces `"error-message":"null"` with a diagnostic string before notifying holon. By monitoring both the agent's inputs and outputs, the wrapper can reliably update holon on whether the agent successfully met its completion condition or encountered a failure partway through.

Figure 5.4 illustrates the communication flow within an agent, detailing its interactions with the holon, channels, and environments. Once a communication channel is set up,

the agent wrapper is initialized upon request from the holon. When the agent wrapper is ready, it notifies the holon via XMPP.

Subsequently, the agent wrapper requests the address of the communication channel or environment it intends to interact with, to which the holon responds. The agent wrapper then initiates communication with the corresponding channel through XMPP, prompting the channel to establish a connection via either TCP or UDP.

In this scenario, the channel first contacts the agent wrapper with an input message. Upon receiving the input, the agent wrapper forwards it to the agent through the appropriate communication channel. The agent then processes the input, and once it is ready to respond, the agent wrapper receives the output via the output channel. This setup allows for different protocols to be used for agent input and output, but it can make handling more complex, especially when the communication protocol is designed to be synchronous, such as HTTP. Depending on the communication flow specification, the agent wrapper may forward the output to other channels or environments as needed. Finally, once the agent completes its processing, it acknowledges completion to the holon via XMPP.

Figure 5.4 presents a sequence diagram illustrating agent communication with other entities within the MAS.

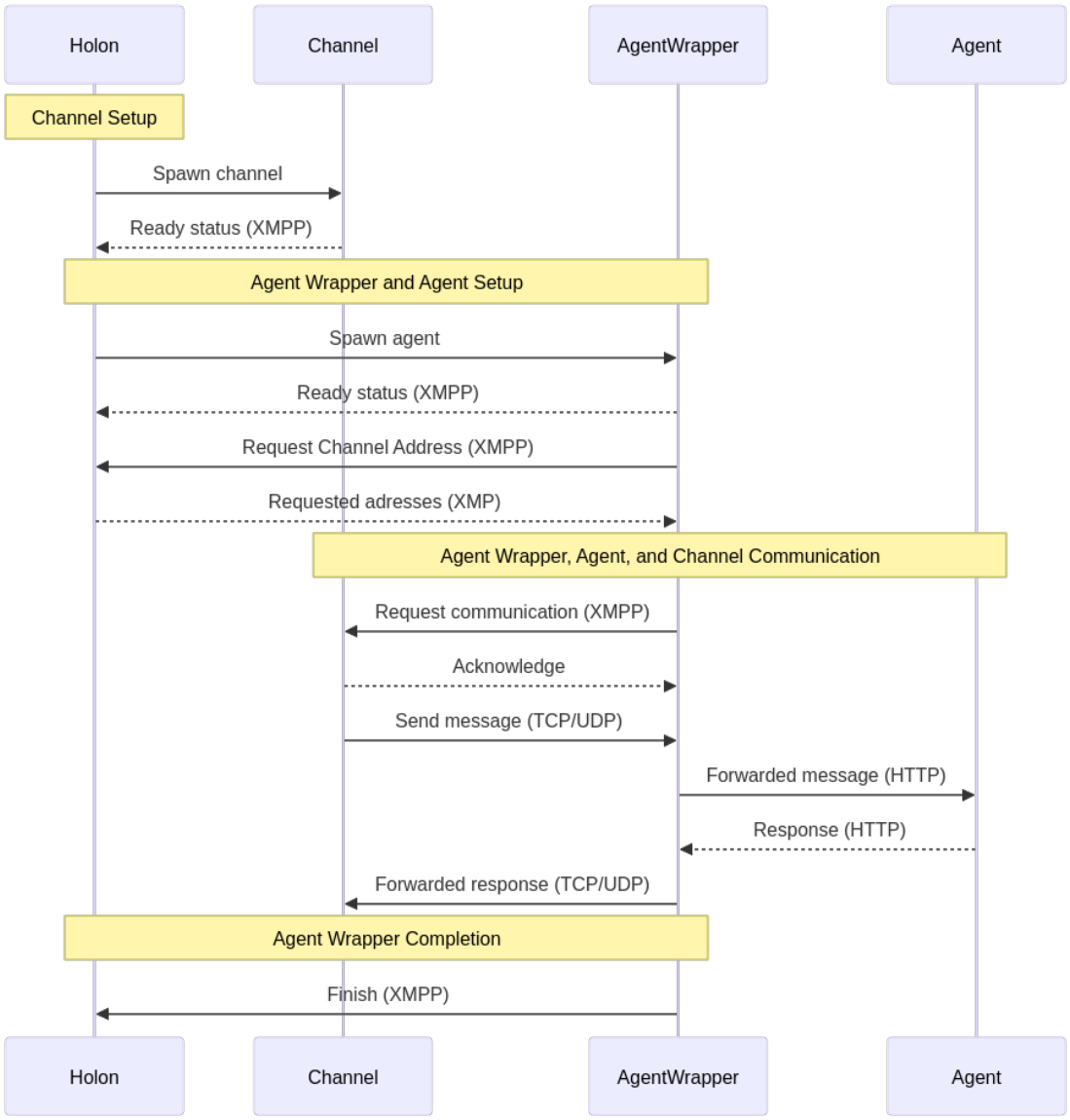

Figure 5.4: Agent communication with other entities within MAS

5.2.5.1 Agent definition

Agent definitions are written in YAML and define how to interact with an agent. These specifications are consumed by agent wrapper to configure the socket, which serves as the communication bridge between the platform and the agent.

Each agent definition is stored in a file with the `.ad` extension. The filename must exactly match the agent's name to ensure consistency with the name used in the communication flow specification. This consistency is essential for the orchestration platform to correctly identify and connect with the agent.

When the orchestration platform processes an orchestration specification and encounters an agent named `client_agent`, it searches for a corresponding definition file named `client_agent.ad`. If the file is missing or misnamed, the platform will fail to establish

communication with the agent, potentially causing orchestration errors.

The agent definition consists of the following properties:

- `name`: The unique identifier for the agent, ensuring it aligns with the name used in the agents' orchestration specification.
- `description`: A free-text field describing the purpose and functionality of the agent.
- `start`: Command to execute in order to start the agent.
- type: Indicates the startup mode of the agent. Supported values are:
  - `UNIX`: The agent runs as a UNIX terminal process [29].
  - `DOCKER`: The agent is started as a Docker container [110].
  - `Kubernetes`: The agent is started as a Kubernetes container [18].
- `input`: Defines how the agent receives data and the characteristics of the input.
  - `type`: Specifies the communication protocol used to send data to the agent. Available values:
    - `STDIN`: Data is provided through standard input.
    - `FILE`: Data is read from a file.
    - HTTP: Data is received via an HTTP request.
    - WS: Data is received over a WS connection.
    - `NETCAT`: Data is streamed using Netcat.
  - `data-type`: Defines whether the input consists of a single value or a continuous stream. Available values:
    - `STREAM`: The input arrives as a continuous stream of data.
    - `ONEVALUE`: The input is a single discrete value.
  - `fmt`: Specifies the format of incoming data. This is especially relevant when data arrives in chunks, as the received chunk may not represent a complete value.
  - `cutoff`: Determines how the agent identifies separate data chunks before processing the message. Available values:
    - `DELIMITER`: A predefined character or sequence separates chunks.
    - `TIME`: Data is processed at time-based intervals.
    - `SIZE`: Data is processed in fixed-size chunks.
    - REGEX: A REGEX pattern is used to detect chunk boundaries.

  - `end`: Relevant for streaming input, this property defines when a message is considered complete.
  - `value-type`: Specifies the data type of the incoming value. Available values:
    * `string`: The input is a text-based value.
    * `binary`: The input consists of binary data.

- `output`: Defines how the agent sends data, with the same configuration options as input.

As shown in the above description, the agent definition specifies configurations for input and output separately. This separation may not be necessary for synchronous, request-response communication protocols such as HTTP. However, the rationale behind defining them separately is twofold: it allows for different communication protocols to be used for input and output, and it supports protocols that are not based on a request-response model, which require distinct interfaces for input and output communication.

The following are two examples of agent definitions: one that communicates via HTTP with a single value, and another that communicates via WS using streaming.

```
agent:
  name: http_onevalue_agent
  description: >
    An agent that receives a single value via HTTP
    and processes it as a one-time request.
  type: unix
  start: python3 -m http_server
  input:
    type: HTTP http://localhost:5000/input
    data-type: ONEVALUE
    value-type: STRING
  output:
    type: HTTP http://localhost:5000/output
    data-type: ONEVALUE
    value-type: STRING
```

Listing 5.50: HTTP-based agent

```
agent:
  name: stream_ws_agent
  description: >
    An agent that streams data via WebSocket.
    It continuously receives and sends data.
  type: unix
  start: python3 -m ws_server
  input:
    type: WS ws://localhost:8080
```

```
10      data-type: STREAM
11      fmt: { "data": DATA }
12      cutoff: DELIMITER \n
13      end: <!eof!>
14      value-type: STRING
15    output:
16      type: WS ws://localhost:8090
17      data-type: STREAM
18      fmt: { "data": DATA }
19      cutoff: DELIMITER \n
20      end: <!eof!>
21      value-type: STRING
```

Listing 5.51: WS-based agent

5.3 Tutorial

The following is a high-level, step-by-step guide for migrating an existing agentic system to the artifact. These same steps also apply when developing a new system intended for use with the artifact. This guide is intended to help users better understand the migration requirements and the necessary steps involved. At a high level, the process consists of writing specifications and adjusting the agents' communication contracts to ensure seamless integration.

To integrate the artifact into this system, the steps would be as follows:

1. Adjust agent communication protocols and contracts
    - If necessary, modify the communication protocols and agent contracts to align with the supported protocols of the orchestration platform.
2. Ensure clear and atomic agent responsibilities
    - Agents should have well-defined responsibilities and be implemented atomically to promote modularity and maintainability. This helps ensure that each agent remains simple, focused, and predictable in its behaviour.
3. Write agent definitions
    - Define specifications outlining how the orchestration platform should interact with each individual agent, ensuring proper communication and coordination.
4. Define required elements for the agents' orchestration specification

- Identify the required elements such as channels, environments, agents, and dependencies that are used to specify communication flows within the agents' orchestration specification.

5. Write agents' orchestration specifications

   - Define the agents, channels, environments, holons, and their dependencies within the agents' orchestration specification.

6. Startup the agent orchestration system

   - Launch the orchestration platform to manage communication and coordination between agents through command-line.

Additional materials, examples, and instructions are available in the public GitHub repository [78]. After the orchestration platform starts successfully, an output similar to that shown in figure 5.5 should appear.

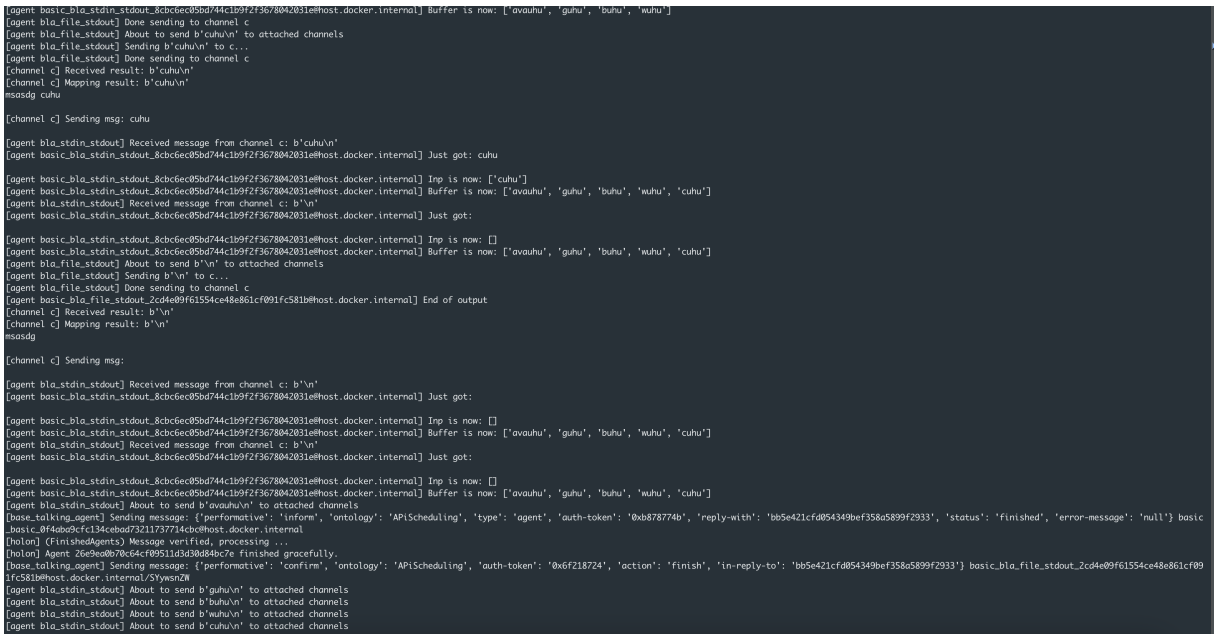

Figure 5.5: Output indicating successful startup of the orchestration platform

Chapter 6

Demonstration

In this chapter, the focus is on validating and testing the integration of the artifact within systems. Specifically, four use cases from the O HAI 4 Games project are explored. The analysis examines ease of integration, scalability, modularity, and other factors, highlighting both strengths and areas for improvement. Three of the use cases involve integrating the artifact into an existing system that has already been developed, showcasing the process of converting an existing system. However, one use case takes a different approach, demonstrating the development of a new system with the artifact.

6.1 MMORPGs

One of the use cases where the artifact has been tested is in game engine layers designed to support Massively Multiplayer Online (MMO) Interactive Fiction (IF) games. This implementation builds upon Inform 7 [87], a declarative programming language used to develop text-based narrative experiences where players interact with the game world through textual commands. Inform 7 typically structures its game environments as interconnected rooms, allowing players to navigate and engage with in-game objects, NPC, and other interactive elements. The challenge addressed in this work is extending Inform 7's single-player narrative-driven gameplay into a multi-player experience where multiple players can coexist and interact in a synchronized virtual world [97].

To achieve this, MAS approach is employed, integrating an additional game engine layer that facilitates player interaction and world synchronization over a network. Communication between players is enabled through the XMPP [93], which allows real-time text-based interactions within the game. This is implemented using the SPADE framework, which provides an agent-based infrastructure for managing communication between players and synchronizing the game state [70]. Players can send messages to individuals or broadcast them to everyone in the same in-game location, mimicking real-world conversations within a narrative environment. The system also notifies users when other players enter their current room, maintaining a sense of presence and immersion. A Py-

thon interface to the Glulxe IF [2] interpreter is used to process player input and track room transitions, ensuring that the game world remains responsive to player actions. This implementation showcases the potential for transforming traditional text-based IF games into dynamic, multi-player experiences while preserving the core mechanics of interactive storytelling [97].

6.1.1 Initial state

In the initial setup, a MAS has been implemented using the SPADE framework. This system consists of a `central` agent, which acts as a server, and multiple `client` agents that communicate with it. The primary role of the `client` agents is to notify the `central` agent about updates, such as changes in their location or messages crafted by their users.

For `client` agents to exchange messages with one another, they must first subscribe to the `central` agent. By subscribing, a `client` agent expresses its interest in receiving messages that are relevant to its context, while also indicating that it will be sending out its own messages based on actions it performs. When a `client` agent sends a message, the `central` agent determines which subscribed agents should receive it and forwards the message accordingly.

Developed on the SPADE framework, the system provides adequate coordination, allowing agents to interact dynamically through a `central` agent. The architecture is designed to be scalable and can theoretically support any number of agents.

6.1.2 Transformed state

In the transformed state, a `central` agent is responsible for consuming all activity messages from `client` agents. `client` agents, as before, continue to share their state changes by publishing activities to `activity` channel. `central` agent listens to these messages and processes them accordingly.

In this setup, communication occurs over TCP, ensuring that every message sent by a `client` agent is reliably processed. `central` agent applies additional processing to `activity` channel messages, distinguishing between interaction messages and location changes. It then forwards these messages to two separate channels: `messages` and `location_changes`. This allows `client` agents to selectively subscribe to the type of messages they are interested in (for example, a user may choose to listen only to interaction messages while ignoring location changes).

Migrating from the initial state to the transformed state required several changes. The initial implementation was built on the SPADE framework, necessitating efforts to replace its communication layer and decouple both `central` and `client` agents from SPADE. Instead, agents were restructured to communicate via WS protocol. Additionally, minor adjustments were made to the business logic of both `central` and `client` agents to

ensure proper message processing based on the designated channels (`location_changes` or `messages`). The execution flow specifies that `central` agent starts up first, after which the client agents start, to make sure no activity events are sent before. Below is agents' orchestration specification for this use case:

```
// channels
channel activity

channel messages

channel location_changes

// agents
agent client_1:
  self -> activity
  messages -> self

agent client_2:
  self -> activity
  messages -> self

agent central:
  activity -> self
  self -> messages
  self -> location_changes

// execution flows
start central client_1 | client_2
```

Listing 6.1: Agents' orchestration specification for MMORPG use case

The following is an agent definition for `central` and `client` agents, which differ only in the host, port they connect to, script they run, name, and description:

```
agent:
 name: central
 description: >
   Central agent used for communication in MMORPG IF.
 type: unix
 start: central.py
 input:
  type: NETCAT localhost:3000
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
```

```
14  output:
15   type: NETCAT localhost:4000
16   data-type: STREAM
17   fmt: { "data":DATA }
18   cutoff: DELIMITER NEWLINE
19   end: <!eof!>
20   value-type: STRING
```

Listing 6.2: `central` and `client` agent definition for MMORPG use case

The figure below illustrates the MAS setup derived from the artifact.

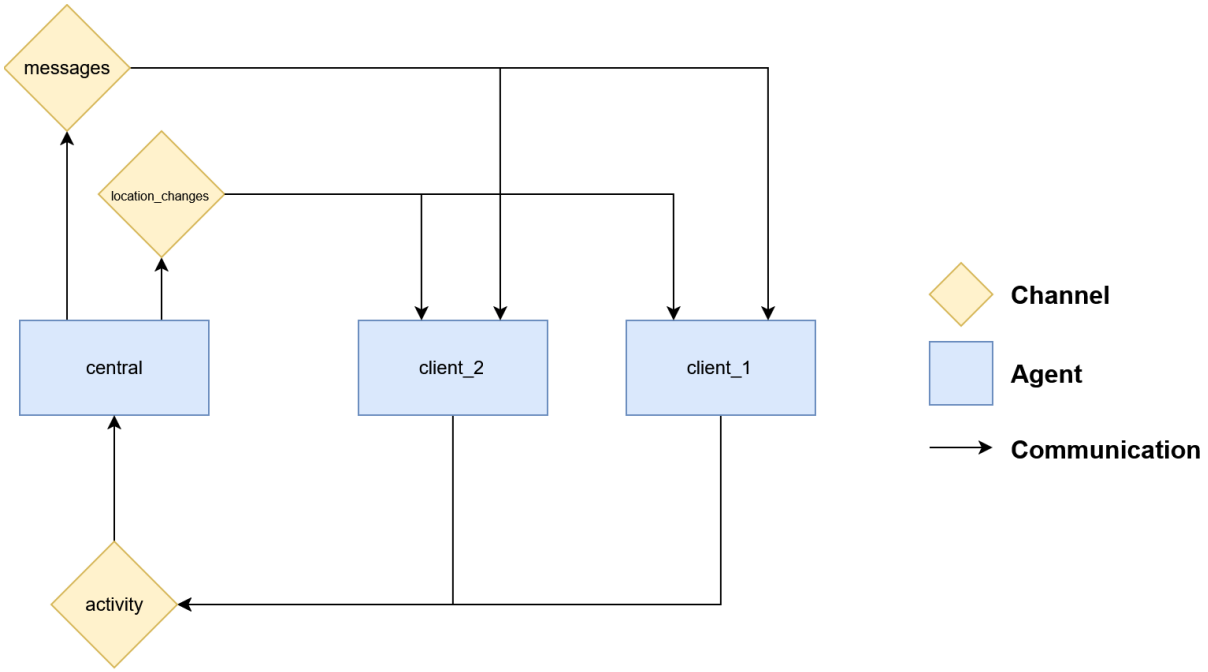

Figure 6.1: Agent architecture for MMORPG use case

6.1.3 Pros and cons

Following is a list of pros and cons identified during the migration of the solution to the artifact:

- Pros
  - Supports high scalability, enabling seamless expansion of agent-based systems.
  - Requires minimal changes to integrate with the artifact, reducing migration overhead.
  - Provides easier configuration for specifying event listeners, simplifying event-driven workflows.

  - Allows agents to be integrated across different machines, whether local or distributed, making it particularly beneficial for MMO environments.

- Cons

  - The existing communication protocol XMPP is not supported, necessitating migration to a compatible alternative.
  - Agents cannot easily subscribe to or modify communication channels at runtime, making it difficult to adapt to dynamic messaging needs (for example, switching location which should then subscribe to the corresponding channel).
  - Agents must be predefined in the communication flow specification, which prevents the addition of new agents at runtime (e.g., new players joining).

6.2 Cognitive agents and gamification

The integration of cognitive agents and gamification into telemedicine presents a novel approach to enhancing patient engagement and healthcare service delivery. At the core of this innovation is Beautiful Artificial Intelligence Cognitive Agent (B.A.R.I.C.A), an AI-driven system that utilizes NLP, ML, Belief-Desire-Intention (BDI) models, and automation to facilitate real-time interactions between patients and medical professionals. By embedding Speech-to-Text (STT) and Text-to-Speech (TTS) capabilities, the system enables seamless communication, making healthcare more accessible and interactive. Additionally, gamification techniques are integrated to encourage adherence to medical protocols, turning essential health-related tasks into engaging, reward-based experiences [99, 95].

The system employs Holonic Multiagent Systems (HMASs) [25], where different microservices act as autonomous agents capable of intelligent decision-making and coordination. It supports multiple stakeholders, including patients, physicians, nurses, and specialists, who can access it via smart devices, PCs, or smart TVs. It is designed for chronic disease management, emergency response assistance, virtual consultations, and preventive healthcare education, with gamification elements such as progress tracking, achievement-based rewards, and interactive avatars [99, 101].

6.2.1 Initial state

B.A.R.I.C.A system has been designed and implemented following a HMAS architecture. This architectural choice was made because the agent is publicly exposed and may be accessed by multiple external parties. Within the holon, several specialized agents collaborate to handle different roles efficiently.

Concretely, the system consists of an agent that processes audio input by transcribing it into text. Another agent applies intelligent business logic to interpret the transcribed inquiry and interact with a knowledge database to generate an appropriate response. Finally, a third agent converts the textual response back into audio output, ensuring a seamless conversational experience.

Communication with the system can occur via either a Representational State Transfer (REST) endpoint or WS. Given that B.A.R.I.C.A is designed for real-time communication, WS technology offers better performance for this use case. Additionally, since communication between agents follows a sequential flow, a custom message-passing protocol has been implemented to optimize inter-agent interactions.

6.2.2 Transformed state

In the transformed state, the new implementation consists of four distinct agents: `user`, `transcriber`, `knowledge`, and `audio_generation`. These agents communicate primarily through dedicated UDP channels, however, in some cases, they rely on HTTP where the loss of a packet is critical.

The user agent captures speech input and sends it via the `audio_input` channel. `transcriber` agent listens to this channel, processes the incoming audio, and generates text transcriptions, which it then transmits through the `transcription` channel. `knowledge` agent consumes transcriptions, processes the input, and generates text-based responses. These responses are then sent to the `response` channel.

The audio generation agent listens to the `response` channel, converts textual responses into audio, and transmits the generated speech via the `audio_output` channel. `user` agent then listens to the `audio_output` channel to receive and play back the synthesized speech.

To support this transition, the communication contracts between agents were adjusted to be more modular and atomic. Despite these changes, the core business logic remains the same, as the fundamental flow of information is preserved. Most communication happens over UDP, minor packet loss is tolerable in specific cases, making it a suitable choice for lightweight, low-latency interactions.

Below is agents' orchestration specification:

```
1  // channels
2  channel transcription
3
4  channel response
5
6  channel audio_input
7
8  channel audio_output
9
10 // agents
```

```
agent transcriber:
  audio_input *-> self
  self *-> transcription

agent knowledge:
  transcription *-> self
  self -> response

agent audio_generation:
  response -> self
  self *-> audio_output

agent user:
  self *-> audio_input
  audio_output *-> self

// execution flows
start transcriber | knowledge | audio_generation | user
```

Listing 6.3: Agents' orchestration specification for cognitive agents and gamification use case

The following is an agent definition for `transcriber` agent, which would be very similar to `user` and `audio_generation` agents, except for differences in name, description, Netcat ports, and start command:

```
agent:
 name: transcriber
 description: >
   Agent responsible for transcription.
 type: unix
 start: transcriber.py
 input:
  type: NETCAT localhost:3002:udp
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
 output:
  type: NETCAT localhost:4002:udp
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
```

Listing 6.4: `transcriber` agent definition for cognitive agents and gamification use case

Below is an agent definition for `knowledge` agent:

```
agent:
 name: knowledge
 description: >
   Agent responsible for generating response to user message.
 type: unix
 start: knowledge_server.py
 input:
  type: HTTP http://localhost:2709/
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
 output:
  type: HTTP http://localhost:2709/response
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
```

Listing 6.5: `knowledge` agent definition for cognitive agents and gamification use case

The figure below illustrates the MAS setup derived from the artifact.

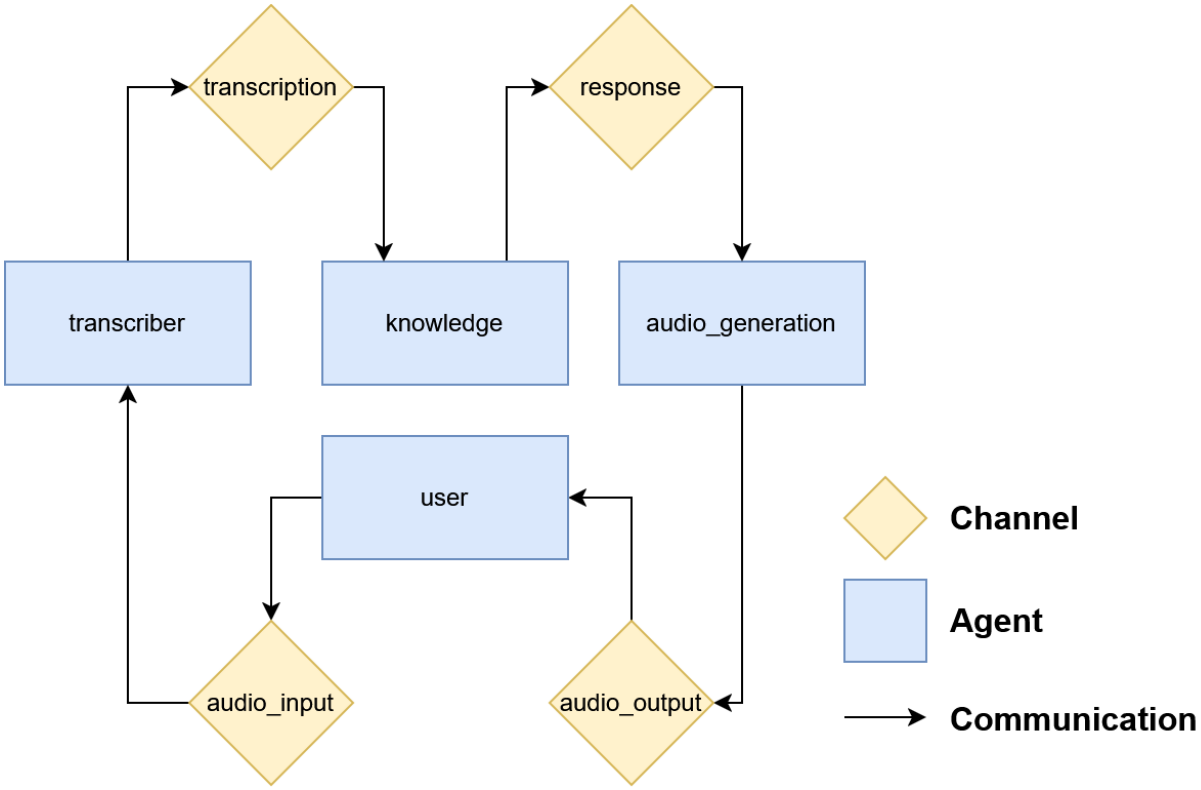

Figure 6.2: Agent architecture for cognitive agents and gamification use case

6.2.3 Pros and cons

Below are the pros and cons of the transformed state.

- Pros
  - Supports UDP for efficient, low-latency streaming communication.
  - Seamless integration with intelligent agents, particularly those accessible via APIs.
  - Requires minimal modifications to existing business logic, ensuring an easy transition.
- Cons
  - The necessity of adding an additional agent, `user` agent, arises from the concept of having channels for communication between the user graphical interface and other components.
  - Using the HTTP protocol for agent communication required separate endpoints for input and output, which is redundant given that HTTP inherently follows a request-response communication model.

6.3 AVs – serious gaming

The use case of Digital Twins (DTs) in AVs is primarily designed to enhance the safety, performance, and adaptability of self-driving systems. A DT [52] is a real-time virtual representation of an AV, integrating live sensor data, simulations, and ML-based decision-making. Unlike traditional simulations, DTs continuously mirror the real-world environment and vehicle state, providing a dynamic and accurate reflection of physical conditions. In autonomous driving, this concept is leveraged to predict vehicle behavior, optimize traffic management, and enhance cybersecurity by detecting anomalies before they cause disruptions. Additionally, DTs assist in propulsion management, battery optimization, and accident prevention by running virtual test scenarios and refining vehicle responses in real-time [98].

To enable real-time monitoring, intervention, and adaptability in AVs, the DT acts as a high-fidelity virtual counterpart of the physical system. It continuously synchronizes with real-world vehicle data, processing sensor inputs, actuator outputs, and decision-making processes to mirror and predict behavior accurately. By structuring the DT as a game actor, this approach allows for interaction within a simulated environment, enabling developers to test AI-driven optimizations, run safety-critical scenarios, and refine vehicle responses in a controlled yet realistic setting. This setup ensures parallel execution of both the physical vehicle and its DT, allowing real-time feedback loops where insights from the virtual model can immediately inform real-world operations [52, 98].

Moreover, this use case has been designed with the application of the artifact-based framework from scratch, thus there was no need for migrating the code to a new infrastructure.

6.3.1 Initial state

The solution has been designed with the artifact in mind from the outset, ensuring seamless integration and scalability. Each AV is conceptualized as a holon, a self-contained and autonomous unit that can interact dynamically within a larger system. The holon is composed of four distinct agents: `sensor` agent, `external_service` agent, `actuator` agent, and `inter_vehicle` agent, each fulfilling a crucial role in the vehicle's decision-making and operational framework [98].

`sensor` agent represents a variety of onboard sensors responsible for detecting critical parameters such as speed, braking force, environmental conditions, and other vehicle dynamics. These sensors continuously capture and transmit real-time data to the DT, enabling precise situational awareness [98].

`external_service` agent functions as an intermediary between the vehicle and globally accessible services, such as Global Positioning System (GPS), cloud-based mapping systems, and traffic monitoring networks. This agent collects external data that is essen-

tial for navigation, route optimization, and real-time traffic adaptation, ensuring that the vehicle operates efficiently within its surroundings [98].

`actuator` agent is responsible for executing control commands based on the processed data. It determines the appropriate energy distribution for propulsion, braking, and steering, thereby optimizing vehicle performance and ensuring safe operations [98].

`inter_vehicle` agent is a dedicated component for analyzing and integrating data received from other AVs. It processes shared motion and behavioral data, allowing the vehicle to anticipate nearby movements, adapt to dynamic road conditions, and enhance cooperative driving strategies. By aggregating and interpreting this information, the agent facilitates swarm intelligence, enabling real-time collaborative decision-making among autonomous vehicles [98].

The `sensor`, `inter_vehicle`, and `external_service` agents communicate with the `actuator` agent via the `insights` channel. To optimize data transmission, the system leverages UDP and TCP protocols based on data priority:

- UDP is used for high-frequency, non-critical data from sensors and external services (e.g., speed readings, GPS). This ensures minimal latency and real-time responsiveness.
- TCP handles critical information such as `inter_vehicle` communication and actuator commands, ensuring reliable, ordered, and lossless data delivery for safety-sensitive operations.

The holon operates within a structured communication framework, with defined input and output environments. The input environment gathers motion-related data from other holons (i.e., other AVs within the MAS) allowing for coordinated traffic flow and predictive driving. Conversely, the output environment enables the AV to communicate its own motion intentions, ensuring that surrounding vehicles can anticipate and react to its actions. This interconnected exchange of information fosters a decentralized, cooperative driving ecosystem where vehicles dynamically adjust their behavior to enhance safety, efficiency, and overall traffic management [98].

By structuring AVs as holons with dedicated processing agents and an optimized communication protocol strategy, this approach not only enhances modularity and scalability but also aligns with the DT paradigm, reinforcing the system's ability to perform real-time simulations, optimizations, and predictive analysis while maintaining low latency for frequent updates and high reliability for critical information processing [98].

Below is the agents' orchestration specification for a single AV holon:

```
1 // holons
2 import holonAV2
3
4 // environment
```

```
environment .

// channels
channel insights

// agents
agent inter_vehicle:
  ENV_INPUT *-> self
  self -> insights

agent sensor:
  self *-> insights

agent external_service:
  self *-> insights

agent actuator:
  insights -> self
  self -> ENV_OUTPUT

// execution flows
start inter_vehicle | sensor | external_service | actuator
```

Listing 6.6: Agents' orchestration specification for AVs – serious gaming use case

The following is an agent definition for `sensor` agent. However, `inter_vehicle`, `external_service`, and `actuator` agents would follow the same structure, except for differences in host, port, protocol, name, description, and run command values:

```
agent:
 name: sensor
 description: >
   Agent responsible for collecting various analytical data of a vehicle
      .
 type: unix
 start: sensor.py
 input:
  type: NETCAT localhost:3003:udp
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
 output:
  type: NETCAT localhost:4003:udp
  data-type: STREAM
  fmt: { "data":DATA }
```

```
18   cutoff: DELIMITER NEWLINE
19   end: <!eof!>
20   value-type: STRING
```

Listing 6.7: `sensor` agent definition for AVs – serious gaming use case

The figure below illustrates the MAS setup derived from the artifact.

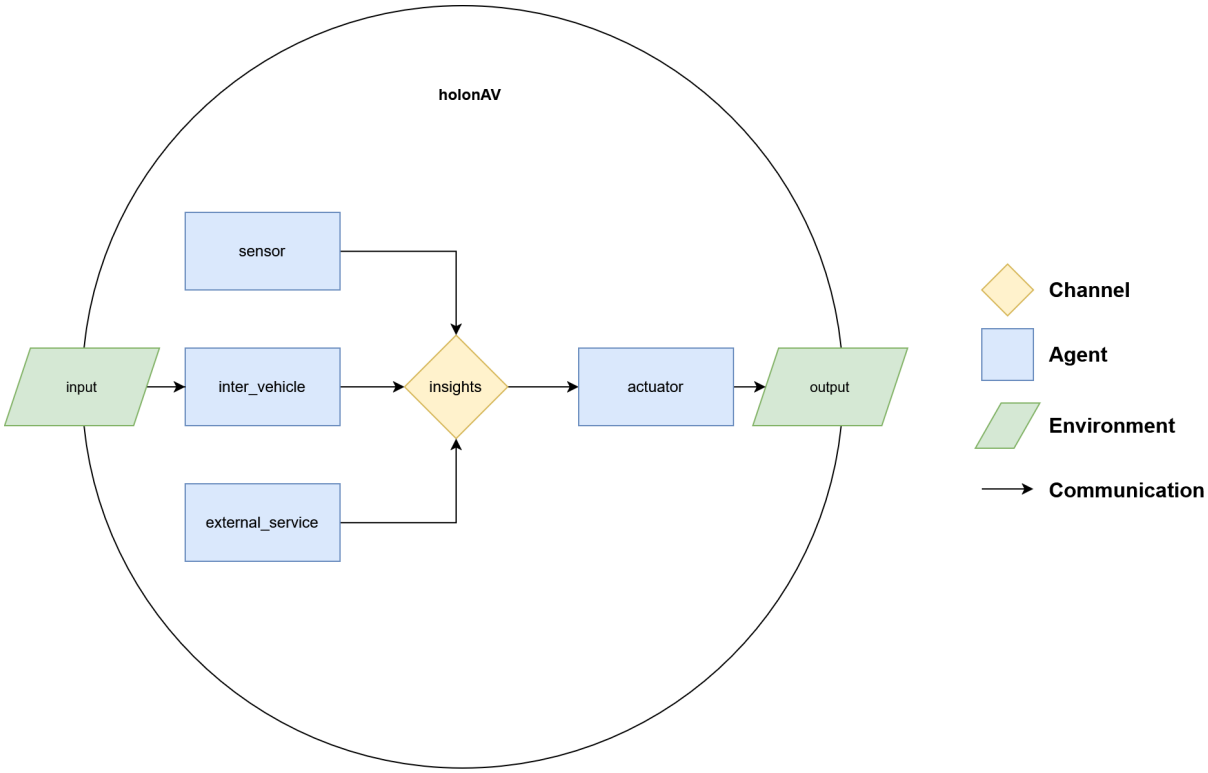

Figure 6.3: Agent architecture for AVs – serious gaming use case

6.3.2 Pros and cons

Below are the pros and cons of the transformed state.

- Pros
  - Provides efficient, low-latency streaming communication through UDP.
  - Enables a holonic architecture, which is highly valuable for building clusters of agents that communicate with one another.
  - Facilitates the development of new systems that align with the artifact's design expectations.
- Cons

- Communication between agents (via channels) is limited to TCP or UDP, restricting protocol flexibility.
- Holonic environments can only be accessed by other holons initiated within the orchestration platform, preventing interaction with external agents.

6.4 Game streaming system

Another use case developed as part of O HAI 4 Games is Lag but Good Game (laGGer), a game streaming engine designed entirely using open-source technologies. The system enables real-time game streaming over the cloud, removing the need for high-end hardware on the client side. Unlike proprietary commercial platforms such as Google Stadia [37] or Nvidia GeForce Now [63], laGGer emphasizes accessibility and flexibility by leveraging open standards and microservice-based architectures. The core innovation is its integration of MASs, which orchestrate multiple game instances and enable seamless video and audio streaming. This design supports real-time simulations, interactive gaming experiences, and collaborative virtual environments, making it a versatile tool beyond entertainment, particularly for transport and mobility research [57].

To achieve this, laGGer utilizes a cloud-based infrastructure that allows multiple users to interact with game environments remotely. It is designed to be highly scalable and adaptable, supporting different transport and mobility research applications, such as traffic simulations, AV testing, and public transportation modeling. The system enables real-time data collection and analysis, providing valuable insights into user behavior and mobility patterns. Additionally, its collaborative capabilities facilitate multi-player participation, making it useful for training scenarios and decision-making simulations in urban planning [57].

6.4.1 Initial state

The laGGer system follows a MAS architecture, where key components operate as autonomous agents that communicate asynchronously. `game_streaming` agent orchestrates the entire system, managing game sessions and assigning `game` agents, which run inside Docker containers to ensure isolation and scalability. Each `game` agent executes a specific game environment and streams it to users via noVNC and X11 forwarding, allowing access without requiring additional installations. `videoconferencing` agent facilitates real-time communication using WebRTC, enabling users to interact within shared simulations [57].

While only `game` agents are containerized for modularity, the other agents (`game_streaming` agent and `videoconferencing` agent) run as standalone processes, leveraging SPADE and XMPP for efficient coordination. When a user initiates a session, `game_streaming` agent assigns an available `game` agent, which reports status updates during streaming. This

architecture allows laGGer to deliver a flexible, scalable game-streaming solution, supporting transport and mobility research applications such as traffic modeling, AV testing, and urban planning simulations [57].

6.4.2 Transformed state

In the transformed state, assumption is that each server or pool of players who either play together or are in some way related forms a single holon. A holon consists of multiple `game_instance` agents, each responsible for running a game in Docker, which the user interacts with indirectly. When a user decides to play a game, they interact with the `user` agent, which captures their actions and delivers the video feed produced by `video_conferencing` agent. `user` agent sends messages to `user_actions` channel, which `game_orchestrator` agent listens to for incoming events. When an event arrives, `game_orchestrator` agent communicates with the appropriate `game_instance` agent via `game_actions` channel. `game_orchestrator` agent keeps track of active game instances, their states, and how they map to users. `game_instance` agent reports state changes and game outputs through the `game_events` channel. `orchestrator_agent` listens to this channel to determine whether a game session is still in progress or has been completed.

`video_conferencing` agent also listens to the `game_events` channel to process the game instance's output and convert it into a video feed. `video_conferencing` agent then publishes this feed to the `video_output` channel, which `user` agent subscribes to. `user` agent delivers the feed to the graphical interface, allowing the user to see the game's output. Communication between agents occurs over the UDP protocol, prioritizing low latency over reliability since minor packet loss is acceptable in exchange for real-time responsiveness. In this transformed state, there are four primary communication channels: `user_actions`, `game_events`, `video_output`, and `game_actions`. Since multiple users and game instances exist at any given time, agents may receive messages that are not relevant to them. Each agent reads message metadata to filter out any that do not pertain to its assigned task.

The majority of the agent codebases for each individual agent remained unchanged, though some effort was required to update the communication contracts between them. Additionally, a new agent, `user` agent, was introduced, which was not part of the initial state but is now required to facilitate user interactions and manage video delivery.

Below is agents' orchestration specification:

```
1
2 channel user_actions
3
4 channel game_actions
5
6 channel game_events
```

```
7
8  channel video_output
9
10 // agents
11 agent game_orchestrator:
12   user_actions *-> self
13   game_events *-> self
14   self *-> game_actions
15
16 agent video_conferencing:
17   game_events *-> self
18   self *-> video_output
19
20 agent game_instance_1:
21   user_actions *-> self
22   self *-> game_events
23
24 agent game_instance_2:
25   user_actions *-> self
26   self *-> game_events
27
28 agent user_1:
29   self *-> user_actions
30   video_output *-> self
31
32 agent user_2:
33   self *-> user_actions
34   video_output *-> self
35
36 // execution flows
37 start game_instance_1 game_instance_2 video_conferencing
     game_orchestrator user_1 user_2
```

Listing 6.8: Agents' orchestration specification for game streaming system use case

Following is an agent definition for `game_instance` agent:

```
1 agent:
2  name: game_instance
3  description: >
4    Agent responsible for orchestrating execution between users and game
       instances.
5  type: docker
6  start: game_instance.py
7  input:
8   type: NETCAT localhost:3004:udp
9   data-type: STREAM
```

```
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
 output:
  type: NETCAT localhost:4004:udp
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
```

Listing 6.9: `game_instance` agent definition for game streaming system use case

Below is the agent definition for `game_orchestrator` agent. A similar definition would be used for other agents, differing only in the Netcat host, port, script, name, and description:

```
agent:
 name: game_orchestrator
 description: >
   Agent responsible for orchestrating execution between users and game
      instances.
 type: unix
 start: game_orchestrator.py
 input:
  type: NETCAT localhost:3005:udp
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
 output:
  type: NETCAT localhost:4005:udp
  data-type: STREAM
  fmt: { "data":DATA }
  cutoff: DELIMITER NEWLINE
  end: <!eof!>
  value-type: STRING
```

Listing 6.10: `game_orchestrator` agent definition for game streaming system use case

The figure below illustrates the MAS setup derived from the artifact.

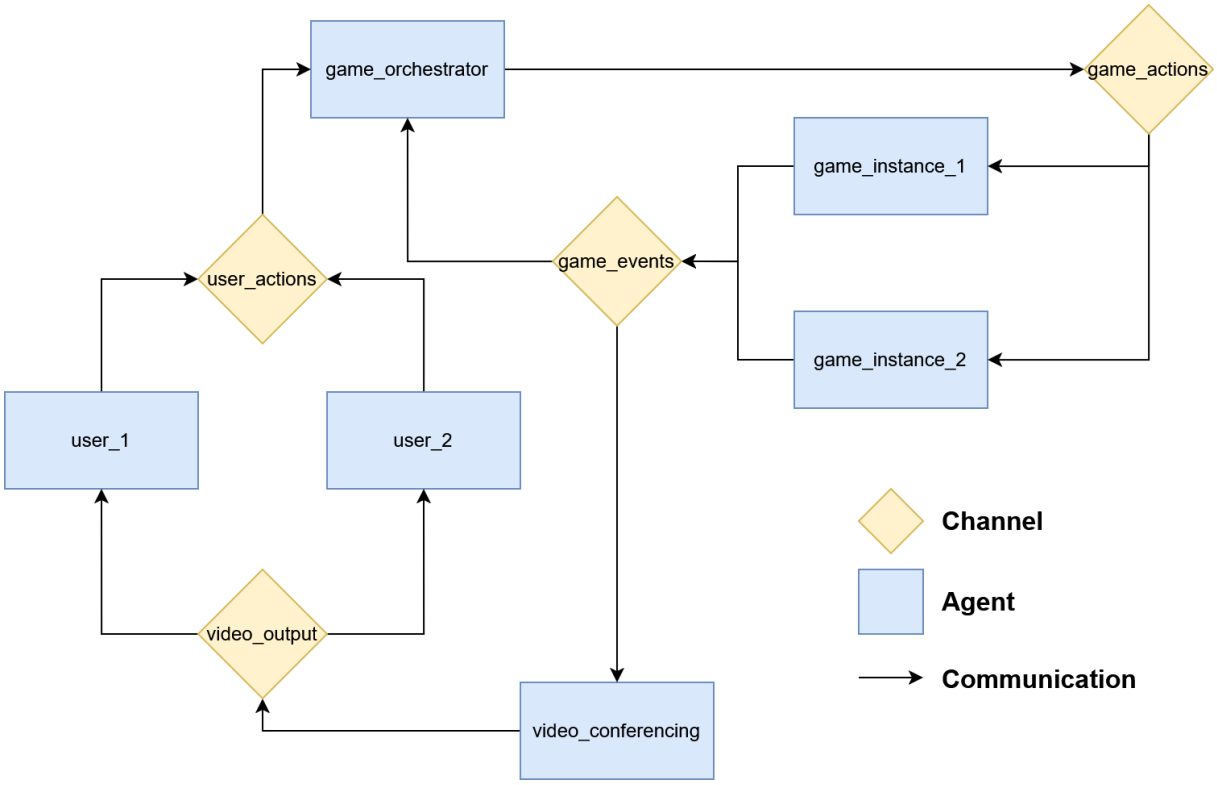

Figure 6.4: Agent architecture for game streaming system use case

6.4.3 Pros and cons

Below is a list of pros and cons identified during the migration process.

- Pros
  - Supports UDP, enabling low-latency communication.
  - Can scale to a high number of agents.
  - Supports Docker for containerized deployment of agents.
- Cons
  - All `game_instance` and `user` agents receive the same output, even if it is not intended for them, requiring additional filtering logic.
  - Dynamic agent creation is not possible as all agents must be predefined and operational.
  - Execution flows are primarily based on agent start and end events, with limited support for other triggers such as occupancy and bandwidth, which would be helpful for proper load balancing.

– Agents must be started through the orchestration platform, which does not allow connecting with already running agents.

Chapter 7

Evaluation

The artifact has been evaluated on its ability to ensure the programming language is equipped with comprehensive features to manage the communication flows identified in the previous phases of research, confirmed through semantic analysis, and further assessed by the O HAI 4 Games project team and independent experts. The evaluation process applies the Framework for Evaluation in Design Science research (FEDS) evaluation framework, which is comprised of four steps: (1) Explicate the goals of the evaluation, (2) Choose the evaluation strategy or strategies, (3) Determine the properties to evaluate, and (4) Design the individual evaluation episode(s) [116].

7.1 Evaluation process

The following section provides a detailed explanation of each step within the FEDS framework, highlighting how these steps are integrated into the design and execution of this research [116].

7.1.1 Explicating the goals of the evaluation

The primary goal of the evaluation was to assess the software artifact's ability to ensure that the programming language is equipped with comprehensive features to manage communication flows, as identified in prior research phases. This has been confirmed through semantic analysis and qualitatively assessed by members of the O HAI 4 Games project team, as well as independent experts with extensive experience in cloud systems, AI, and microservices architectures. The experts hold senior titles and have previously contributed to the design and architecture of projects utilizing MAS architecture.

This evaluation measured how well the artifact integrates into existing software systems, supports modularity and reuse, meets specified requirements across four practical use cases defined by a set of requirements and constraints reflecting real-world conditions, and adheres to these standard programming language criteria.

7.1.2 Choosing the evaluation strategy

Given that the artifact is intended as an extension or replacement for existing agents' orchestration tools in MASs, the evaluation followed the Technical Risk & Efficacy strategy. This strategy ensures thorough technical assessment through a combination of formative and summative evaluations, confirming the artifact's reliability and suitability in replacing current systems [116].

7.1.2.1 Formative evaluation

Artificial (controlled environment) testing was initially performed in a controlled setting using simulations to iteratively identify and resolve technical issues. Project team members provided feedback through structured questionnaires and direct observations. This feedback was qualitatively analyzed to uncover patterns and refine the artifact's performance and reliability, ensuring its suitability for extending or replacing existing tools in information systems [116].

7.1.2.2 Summative evaluation

Naturalistic (real-world environment) testing was conducted through practical implementation in real-world scenarios to assess how well the artifact integrated into existing systems. Feedback from the project team and domain experts was gathered through observations, interviews, and surveys. The collected data was analyzed using qualitative methods to identify trends in practical application, confirming the artifact's efficacy and suitability across real-world use cases [116].

7.1.3 Determining the properties to evaluate

The evaluation process focused on two main groups of properties: artifact quality characteristics derived from the `ISO/IEC 25002:2024` standard [84, 111] and programming language criteria sourced from Programming Language Pragmatics [114].

7.1.3.1 Artifact quality

- Interoperability (sub-characteristic of Compatibility): Evaluates how well the artifact integrates into existing software systems, including data exchange protocols, communication APIs, and ease of integration with external systems.

- Modularity and reusability (sub-characteristics of Maintainability): Assessed through structural analysis, identifying modular design and ease of extending or reusing the system's functionality in other projects.

- Usability: Evaluated based on appropriateness recognizability, learnability, operability, and user interface aesthetics to determine how intuitive and user-friendly the artifact is.

- Performance efficiency: Assessed by examining efficient coding practices, minimizing redundancy, and optimizing resource utilization.

- Functional suitability: Compared against specified requirements to gauge completeness, correctness, and appropriateness for the intended purpose.

7.1.3.2 Standard programming language criteria [114]

- Readability: Evaluated by examining syntax clarity, consistency in naming conventions, and overall code understandability.

- Writability: Assessed by how expressively the language enables programmers to implement functionality, focusing on features that simplify coding.

- Reliability: Measured by error-handling capabilities, type safety, and mechanisms for writing robust, error-free code.

7.1.4 Designing the individual evaluation episode

The evaluation episode is defined as follows:

1. Implementation: Implement the artifact in each of the assigned use cases, documenting the process and challenges encountered.

2. Feedback Collection: Collect feedback through structured questionnaires and interviews, addressing specific characteristics being evaluated.

3. Analysis: Analyze the feedback qualitatively using thematic analysis [62] to identify patterns, common issues, and strengths.

4. Reporting: Generate a report summarizing the feedback and offering recommendations for improvement.

7.2 Evaluators

The feedback was collected from three O HAI 4 Games project members and six independent experts with extensive experience in building microservices and MAS.

7.2.1 O HAI 4 Games project team members

The members of the O HAI 4 Games project team who evaluated the artifact were:

- Bogdan Okreša Đurić is an AI expert currently teaching at the Faculty of Organization and Informatics in Varaždin. Bogdan holds a PhD in Computer Science (CS) and has made significant contributions to the agentic space through a variety of publications.

- Igor Tomičić is a security expert who teaches at the Faculty of Organization and Informatics in Varaždin. He holds a PhD in CS and serves as a lead auditor for `ISO 27001` and `22301`. Igor has made numerous contributions in the fields of security and AI through conference talks, scientific publications, industry collaborations, and participation in international research projects.

- Neven Vrček is a professor at the Faculty of Organization and Informatics. In the past, he served as the Dean of the Faculty. His interests span across AVs, Internet of Things (IoT), and AI.

7.2.2 Independent experts

The following is a list of independent experts who evaluated the artifact.

- Richard Abrich is a principal AI engineer with a PhD in CS. He previously co-founded OpenAdaptAI, an agentic framework designed for controlling personal computers. In the past, he has consulted with several AI focused companies, helping them deliver end to end automated experiences.

- Matthias Bayer is a software engineer with a variety of experience working with San Francisco-based startups in the AI and agent-driven space, including companies such as GM Cruise, Labelbox, and Concierge. He has served as a founding member of several teams, where he was responsible for designing and setting up the architecture from scratch.

- Rio Kierkels is a cloud systems expert with a strong background in MLOps. He has spent several years consulting with European companies to enhance the infrastructure and security of large-scale systems. His work often focuses on designing scalable architectures and implementing reliable deployment pipelines.

- Vineet Sinha holds a PhD in Computer Science from MIT. Over the years, he has worked with large companies to drive automation and improve workflow efficiency. He is best known for his role as Head of Strategic Innovation at Salesforce. Most recently, he has been focused on building a startup in the AI space.

- Peter Skvarenina is a ML graduate from The University of Texas at Austin and is currently studying Generative Artificial Intelligence (Gen AI) and Robotics at Stanford University. In the past, he has worked at several enterprise companies, including JetBrains, Nokia, and D-iD.

- Michael van Elk currently serves as Head of AI at Tribe AI. In the past, he co-founded a startup focused on data lineage. He has extensive experience building AI solutions, leveraging both modern technologies such as Gen AI and more traditional algorithmic approaches.

7.3 Questionnaire

The questionnaire includes all relevant questions, each accompanied by a rationale explaining its purpose and the specific characteristics being evaluated. Questions are designed to meet clearly defined objectives, ensuring alignment with the overall evaluation goals and clarifying how the collected data will be used. To support a more structured and targeted assessment, the questionnaire is divided into three semantically distinct components: Agents' orchestration specification, orchestration platform, and agent definition.

7.3.1 Agents' orchestration specification

1. Question: Do you find the agents' orchestration specification easy to write and understand?
   Rationale: Helps us understand whether writing a specification is intuitive and easy to learn.
   Characteristics: Usability, Readability, Writability

2. Question: Does reading a communication flow specification give you a clear understanding of which agents communicate and how?
   Rationale: Determines whether the specification provides a clear, holistic understanding of agent interactions.
   Characteristics: Readability, Usability, Functional Suitability

3. Question: Do you think the programming language is type-safe enough for reliable implementation?
   Rationale: Assesses whether users encountered edge cases that were not properly safeguarded by the language.
   Characteristics: Reliability, Usability, Functional Suitability

4. Question: Are the semantics of the language comprehensive enough to address real-world use cases?

Rationale: Provides insights into whether the semantics for communication flow specification are sufficient, or if additional constructs are needed.
Characteristics: Functional Suitability, Usability, Readability

5. Question: Are the supported element types (agent, channel, environment, holon, and execution flow) intuitive and descriptive enough?
   Rationale: Assesses whether the breakdown of element types is adequate and intuitive.
   Characteristics: Usability, Readability, Functional Suitability

6. Question: Is the parametrization of agents useful in your use cases?
   Rationale: Provides information on the usefulness of agent parametrization.
   Characteristics: Modularity and Reusability, Usability, Functional Suitability

7. Question: Which of the implemented execution flows do you find most useful?
   Rationale: Provides insights into the types of execution flows that are most beneficial.
   Characteristics: Functional Suitability, Performance Efficiency, Usability

8. Question: Are there execution flows missing from the current system that you think should be added?
   Rationale: Identifies additional execution flows users might expect.
   Characteristics: Functional Suitability, Modularity and Reusability, Usability

9. Question: Is the range of supported communication protocols (TCP and UDP) sufficient to support communication between agents, environments, and channels?
   Rationale: Rationale: Evaluates whether the TCP and UDP protocols (and underlying mechanics) are adequate for agent communication with environments and channels.
   Characteristics: Interoperability, Functional Suitability, Reliability

10. Question: Do you see the concept of holons as valuable? If so, in what ways?
    Rationale: Explores the usefulness of agent hierarchies and the potential for building multi-layered MAS.
    Characteristics: Modularity and Reusability, Usability, Functional Suitability

11. Question: How straightforward do you find replacing an existing agent with another within the system?
    Rationale: Assesses how modular and interchangeable the agent components are, and whether the system supports easy substitution without extensive reconfiguration.
    Characteristics: Modularity and Reusability, Usability, Reliability

12. Question: How does specifying communication flows in this language compare to other agent frameworks you have used?
    Rationale: Helps assess how easily and clearly communication flows can be specified in this language compared to other agent frameworks.
    Characteristics: Usability, Interoperability, Functional Suitability

13. Question: How useful do you find the channel's ability to transform messages from one format to another during transmission?
    Rationale: Assesses whether message transformation capabilities (e.g., JSON to XML) are useful.
    Characteristics: Functional Suitability, Modularity and Reusability, Usability

14. Question: Are there additional concepts you would find valuable in the language?
    Rationale: Identifies additional concepts or functionalities that could enhance the language.
    Characteristics: Functional Suitability, Usability, Modularity and Reusability

7.3.2 Orchestration platform

1. Question: Do you see the artifact being implementable in the majority of systems?
   Rationale: Helps us assess whether the artifact is adaptable and versatile enough to fit into diverse system architectures and use cases, regardless of the underlying infrastructure or objectives.
   Characteristics: Interoperability, Modularity and Reusability, Usability

2. Question: What is the typical number of agents in your MAS?
   Rationale: Offers an understanding of the scale of MAS typically handled by evaluators, giving insights into the artifact's scalability and ability to manage high agent counts.
   Characteristics: Performance Efficiency, Scalability, Usability

3. Question: Do you think the error handling mechanism covers a good amount of edge cases (e.g., an agent goes down)?
   Rationale: Evaluates the robustness and reliability of the platform by assessing how well it handles critical edge cases and ensures continued operation during failures or unexpected scenarios.
   Characteristics: Reliability, Functional Suitability, Usability

4. Question: Can intelligent agents (e.g., Gen AI-powered) be easily integrated with the platform?
   Rationale: Determines whether the artifact supports integration with state-of-the-art AI technologies, ensuring relevance and compatibility with modern intelligent

agent use cases.
Characteristics: Interoperability, Modularity and Reusability, Functional Suitability

5. Question: Do you see any performance downsides in the solution?
Rationale: Assesses potential bottlenecks or inefficiencies that could affect the platform's performance under typical or high-demand workloads.
Characteristics: Performance Efficiency, Reliability, Usability

6. Question: Is the support for distributed agents sufficient, particularly in terms of communication, synchronization, and fault tolerance?
Rationale: Gauges whether the platform adequately addresses the complexities of managing distributed agents, including communication, synchronization, and fault tolerance.
Characteristics: Interoperability, Performance Efficiency, Reliability

7. Question: Does the cloud support (e.g., running in Docker [110]) match your needs in terms of deployment flexibility, scalability, and compatibility with your existing infrastructure?
Rationale: Validates the artifact's compatibility with cloud environments and containerized solutions, ensuring ease of deployment and scalability in cloud-based systems.
Characteristics: Interoperability, Usability, Modularity and Reusability

8. Question: Are there any additional functionalities of the orchestration platform that you would find useful?
Rationale: Provides insights into potential gaps in the platform's feature set, enabling identification of enhancements that could improve its utility and adoption.
Characteristics: Functional Suitability, Usability, Modularity and Reusability

7.3.3 Agent definition

1. Question: Is the agent definition easy enough to write?
Rationale: Determines whether the system provides a user-friendly mechanism for specifying agents, enabling developers to describe agents without excessive complexity or ambiguity.
Characteristics: Usability, Writability, Readability

2. Question: Is the agent definition clear and accessible to non-technical stakeholders, such as project managers or domain experts?
Rationale: Explores whether the specification format is clear and accessible to non-technical users, promoting collaboration and reducing learning curves.
Characteristics: Readability, Usability, Writability

3. Question: Are the supported properties for agent definition sufficient for your use cases? If not, what is missing?
   Rationale: Evaluates whether the current set of attributes in the agent definition comprehensively captures the necessary agent features, or if additional properties are needed for broader applicability.
   Characteristics: Functional Suitability, Usability, Modularity and Reusability

4. Question: How much effort is required to adjust an existing agent for inclusion into the system based on the artifact?
   Rationale: Evaluates the level of modifications needed to make existing agents compatible with the system, highlighting potential barriers to adoption and integration effort for developers.
   Characteristics: Modularity and Reusability, Usability, Reliability

5. Question: Do updates to your existing agent business logic after its inclusion in the artifact-based MAS require changes to the agents' orchestration specification (particularly, specification of communication flows)?
   Rationale: Assesses whether the system supports independent updates to agent business logic without requiring extensive changes to the communication flows specification, ensuring ease of maintenance and scalability.
   Characteristics: Reliability, Modularity and Reusability, Functional Suitability

6. Question: Which communication protocols do your agents utilize for communication with external systems?
   Rationale: Provides insights into the range of communication protocols used, helping determine whether the system accommodates distributed agents and supports the most commonly used protocols.
   Characteristics: Interoperability, Functional Suitability, Reliability

7.4 Interviews

During the interviews, evaluators provided detailed feedback on their experiences with the artifact. The focus was on understanding what they had attempted to build using the artifact, including the types of communication flows envisioned, their goals, and the context in which the artifact was applied. The strengths and limitations of the implementation were examined by exploring the challenges encountered, obstacles that hindered progress, and instances where the artifact did not fully meet expectations.

Feedback was also gathered on potential enhancements and usability improvements. This included identifying additional features or functionalities that could provide value, suggestions for hints or guidelines that might have streamlined the process, and gaps in

documentation or user support that impacted the experience. These insights were used to inform iterative refinements, ensuring better alignment with user needs and facilitating successful adoption for real-world applications.

7.5 Results

The evaluation results were analyzed using thematic analysis. This method is well suited for the qualitative nature of the data, as it supports the identification of patterns across evaluator feedback without discarding nuance. Thematic analysis enables a grounded interpretation of how evaluators engaged with the artifact, capturing not only what worked and what did not, but also why certain experiences emerged. It offers flexibility in working with a structured set of evaluation criteria while remaining open to recurring patterns that may not have been explicitly anticipated [62].

The structure of the analysis mirrors the structure of the questionnaire, which is organized into three sections: agents' orchestration specification, orchestration platform, and agent definition. Within each component, results are organized according to the two evaluation dimensions: artifact quality and standard programming language criteria. Each characteristic under these dimensions serves as an analytical lens. For every characteristic, recurring themes in the data are presented along with representative observations and targeted recommendations. This format maintains a consistent flow while allowing component-specific issues and insights to be surfaced clearly. Together, these findings provide a detailed account of how the artifact was perceived in terms of both its functional role and its underlying design.

7.5.1 Agents' orchestration specification

The key characteristics analyzed in relation to the agents' orchestration specification include interoperability, modularity and reusability, usability, and functional suitability, which fall under the artifact quality dimension. In addition, the analysis considers readability, writability, and reliability, drawn from the standard programming language criteria. Together, these characteristics provide a comprehensive lens for evaluating both the functional properties of the orchestration specification and the expressiveness and robustness of its underlying language.

7.5.1.1 Interoperability

Most evaluators had a positive view of the interoperability offered by the agents' orchestration specification. They felt that the current support for communication protocols like TCP and UDP was enough for the platform's needs. At the same time, some evaluators suggested improvements, especially adding support for more advanced protocols such as

SCTP, MQTT, and gRPC. These could make the platform more compatible with complex distributed systems. A few also pointed out that the system currently lacks built-in support for pub-sub mechanisms and secure protocols, which might be a drawback for certain advanced use cases.

Although the feedback to add more protocols is reasonable, implementing it may require significant changes to the current architecture. At the moment, agents communicate using channels based on UDP and TCP, which are well-suited for this platform because of their simplicity and low overhead. Introducing more advanced protocols like gRPC or MQTT could certainly help address different requirements around security, latency, and scalability. However, these protocols introduce additional complexity and may require a redesign of how agents connect and exchange messages, which falls outside the scope of this research.

Below are key reflections from reviewers:

- "I think the protocols cover the main ones you would expect/want in for this kind of system. The agents are able to choose their protocol, and the channels can translate to the orchestration's internal protocol." from Michael van Elk

- "In its current form I think it is adequate. However the system will probably benefit from adding SCTP as an additional layer 4 protocol. This is a message based protocol with characteristics of both UDP and TCP but with message boundaries built in." from Rio Kierkels

- "XMPP/MQTT/gRPC/MCP would add even more usefulness. XMPP would allow real-time agent to human interaction, MQTT is used by IoT unreliable high-latency networks in automotive/healthcare/transport/financial industries..." from Peter Skvarenina

7.5.1.2 Modularity and reusability

When it comes to modularity and reusability, feedback is largely positive. Evaluators liked the ability to parametrize agents, as this allows the same agent logic to be reused with different communication channels. This flexibility is seen as a major benefit in complex systems. However, some concerns were raised about the lack of validation between the parametrization and implementation, and the optional nature of certain parameters, which could lead to errors. The concept of holons is also viewed favorably, especially for managing complexity in large systems through hierarchical structuring. While holons are seen as useful for encapsulation and logical separation, there are suggestions for improving the abstraction to avoid leaks and to align interfaces more closely with agents. Replacing agents within the system is generally considered easy, provided the new agent adheres to

the same communication interface, although some evaluators expressed concerns about runtime validation and the ease of replacement in production environments.

Improved validation and stronger type safety for agent parametrization could be introduced both at the language level and in how the platform interprets those parameters at runtime. Making agent parametrization mandatory rather than optional adds clarity, as it removes ambiguity and enforces a consistent structure for defining agent behavior. While holons are generally seen in a positive light, they appear to be a less commonly used pattern, which may explain why some users find them less intuitive. In the current implementation, holons and environments are custom designed components that integrate well with the overall architecture. However, aligning their design more closely with widely understood concepts, such as channels, could make them easier to grasp, especially for users who are new to the artifact or unfamiliar with the holonic model.

Key reflections from evaluators are below:

- "(Holons are) conceptually useful for structuring large systems hierarchically and managing complexity via clusters with defined interfaces (environments)..." from Richard Abrich
- "My only concern would be that the parametrization is not validated against the implementation (as the framework aims to be agnostic to this), meaning there may be a mismatch between specification and implementation." from Michael van Elk

7.5.1.3 Usability

Usability of the agents' orchestration specification is viewed in a positive light. Evaluators found it mostly easy to read and write, especially appreciating its concise and minimal syntax. This simplicity is seen as both a strength and a challenge, as it can create a learning curve for new users. The logical and clean structure of the specification helps with understanding, and the syntax is often described as intuitive. However, some evaluators pointed out that the initial experience could be improved, especially when it comes to understanding special keywords and how inputs and outputs are separated. The different entity types are considered clear and descriptive, though a few questioned whether certain concepts, like the environment abstraction, are truly necessary. Compared to other agent frameworks, the orchestration language stands out for its focus on communication flow and its declarative style, rather than mixing logic with traditional code. While this approach is praised for its clarity and simplicity, some evaluators noted that the lack of dynamic features and built-in agent behaviors might make it harder to use in more complex situations.

Comprehensive documentation with a variety of examples could help make learning the programming language easier, especially when it comes to understanding niche keywords and concepts. Adding more explicit type information to the language could also make the

specification easier to read and more intuitive. Visual tools for debugging and exploring the orchestration structure would improve the development experience and make the system more approachable. To enhance flexibility, it could be helpful to support dynamic startup of agents. This might involve introducing new concepts, such as orchestration platform reacting to agent messages or other runtime triggers. However, introducing such dynamic behavior could make it harder to maintain a clear and consistent view of the agents' state within a cluster, which is something the language currently does well and was designed to support.

Following are reflections from the evaluators:

- "It is good and I like it because of simplicity, some of the commercial orchestration frameworks might be more elaborated." from Neven Vrček
- "Yes, (agents' orchestration specification is) easy to write and read." from Vineet Sinha

7.5.1.4 Functional suitability

Most evaluators found the agents' orchestration specification to be functionally well-suited for its intended purpose. They acknowledged its ability to clearly define communication flows between distributed services. The language is praised for covering essential elements like environments, channels, agents, and execution flows, all of which have been shown to handle complex systems such as audio or video streaming and inter-vehicle communication. However, one common concern is the lack of support for dynamic runtime behavior, which limits flexibility. Evaluators expressed interest in features such as dynamic agent life cycle management, more advanced triggering mechanisms, and stronger error handling. While the existing execution flows (especially parallel and sequential) are seen as useful, there is a call for more advanced flows, such as event-triggered or time-based execution. The ability of the channel agent to transform messages is highlighted as a key strength, helping different types of agents work together more effectively.

To build on the existing strengths of the specification, improving error handling and exception management would make the system more reliable and better suited for a variety of real-world scenarios. Expanding support for advanced execution flows, such as event-driven and time-based actions, could give users greater control and flexibility in orchestrating agents. However, this would also require changes to both the orchestration platform architecture and the programming language used to define communication flows, since the current setup only supports predefined configurations. Together, these improvements would make the platform more user friendly and adaptable to a broader range of use cases.

Below are key reflections from reviewers:

- "The parallel execution flow might be the most useful one. Running agents in parallel shows some of the fundamental values of multiagent systems." from Bogdan Okreša Đurić

- "Parallel | for independent tasks, sequential & for pipelines. Error handling ! is crucial for robustness. Restart + good for long-running services." from Richard Abrich

7.5.1.5 Readability

Most evaluators found the agents' orchestration specification to be clear and readable. They described the supported elements (i.e. agent, channel, environment, holon, and execution flow) as intuitive and easy to understand. Many agreed that the roles are well defined and correspond to familiar ideas from MASs or distributed systems, which helps new users get up to speed more quickly. Still, a few pointed out that the difference between the environment and the channel may not be necessary, since their functions appear to overlap. Although the term holon is generally understood, some users noted that it may require extra explanation, especially for those unfamiliar with its origin. When it comes to specifying communication flows, most evaluators praised the clarity and structure, especially in small systems. However, some raised concerns about navigating larger systems, where the growing complexity can make understanding more difficult. Several respondents suggested that tools for visualizing the flow would be helpful.

To improve the experience, it may be worth reconsidering whether the environment and channel should remain as separate entities or be merged to simplify the language model. It is understandable that the agents' orchestration specification becomes harder to read in larger systems, where proper parametrization and possibly additional structuring concepts would be important for maintaining clarity. Introducing a visual tool that can consume the specification and provide a live preview would also be valuable and could likely be integrated smoothly with the programming language.

Key reflections from evaluators are below:

- "...the supported agent types are descriptive and intuitive enough. The names used make it easy and straightforward to determine what the chosen entity type is about and what their role is in the model." from Bogdan Okreša Đurić

- "...it's very easy to reason about this since the file can be read top-down without a lot of jumping back and forth. That's great for building a mental model of the system while reading line by line." from Matthias Bayer

7.5.1.6 Writability

In terms of writability, evaluators found the agents' orchestration specification expressive and logically structured, making it suitable for defining communication flows. However, they also noted that the custom syntax and unique elements like channels and holons introduce a learning curve. Some users pointed out that understanding how to use special keywords such as `ENV_INPUT` is not always straightforward without consulting documentation. While the minimal and domain-specific syntax is seen as a strength, there are concerns about how well it supports more dynamic scenarios, particularly when input and output separation adds complexity to configuration. Additionally, a few evaluators suggested that introducing more verbosity, such as optional type annotations, could improve clarity during development. There were also recommendations to allow writing the specification in formats like YAML, which could make the language more approachable for new users and easier to integrate with existing workflows.

To improve writability, adding more documentation and practical examples would help reduce the learning curve for new users. Aligning some of the keywords with more familiar terms used in other programming languages could also make the specification easier to write and understand. Additionally, Large Language Models (LLMs) could be used to translate natural language descriptions into orchestration specifications, making it easier to get started and providing clear examples for learning. Another helpful tool could be a step-by-step generator, where users can define the number of agents, their communication channels, and other key elements through a guided interface. This would simplify the writing process further and support users in building correct and complete specifications without needing to understand all syntax details from the beginning.

Following are reflections from reviewers:

- "Yes. The specification uses minimalistic syntax that is compact and readable (e.g., agent a : self -> c). However, it has a learning curve due to custom syntax and requires familiarity with concepts like channels and holons." from Igor Tomičić

- "Yes, the configuration is easy to write and read. The limited but domain specific syntax makes it easy to express and understand complex configurations without much noise." from Matthias Bayer

- "It is quite easy to learn, possibly because the language itself is quite small. However I don't completely see why the orchestration spec and the ad need to be in separate languages. Unless if statements and loops are introduced, it might all work from the YAML markdown language." from Rio Kierkels

7.5.1.7 Reliability

Feedback on the reliability of the agents' orchestration specification, and specifically the programming language, particularly regarding type safety, is mixed. Some evaluators reported confidence in the system, noting that they did not encounter edge cases and found the specification easy to follow for maintaining type consistency. However, some expressed concerns about the lack of explicit type checking, which could lead to errors during runtime if the message formats do not match the expectations of the receiving channels. The absence of both static and dynamic type checking was a common concern, with some noting that communication mismatches are only detected when transformation rules fail. This limitation makes static analysis more difficult and reduces clarity when trying to understand the system's behavior.

To improve reliability, it would be valuable to introduce stronger type checking, both during development and at runtime. Adding basic typing features, such as message schemas or type definitions for channels, could help catch mismatches early and make the system more predictable. These additions would also guide developers in using the correct formats and reduce the risk of miscommunication between agents. In addition, providing support for validating the specification before running the orchestration platform would create a quicker feedback loop, helping users identify and fix issues early. Extensions for development environments that can analyze the written specification and check it against the programming language could further improve this experience by highlighting mistakes as the user writes.

Some of reviewer reflections are as follows:

- "Lack of explicit type checking could lead to runtime errors if message formats don't match channel expectations, despite transformation capabilities." from Richard Abrich
- "...there seems to be no static or dynamic type checking. Communication mismatches (e.g., sending XML to a JSON listener) are not flagged unless caught in transformation rules? Introducing basic typing (e.g., schemas) might improve robustness." from Igor Tomičić

7.5.2 Orchestration platform

For the orchestration platform component, the analysis primarily focuses on characteristics from the artifact quality dimension. These include interoperability, modularity and reusability, usability, performance efficiency, and functional suitability. This set of characteristics captures both the structural and operational aspects of the platform, emphasizing how well it supports integration, maintainability, user interaction, and performance in realistic deployment scenarios.

7.5.2.1 Interoperability

Evaluators widely view the platform as strong in terms of interoperability. They recognize its flexibility and adaptability when integrating with various systems, especially those involving distributed services and cloud native applications. Many see the platform as suitable for a wide range of use cases, but some concerns remain about integration with older legacy systems and the potential overhead in production environments. The platform's ability to support intelligent agents is also well regarded, with most evaluators agreeing it can work effectively with Gen AI agents. Distributed agent support is considered one of the platform's key strengths, though some suggested the addition of a discovery service and support for mesh-style communication. Cloud support through Docker [110] and Kubernetes aligns with industry practices and is appreciated, though some users expressed interest in a fully managed or hosted cloud solution to reduce operational burden.

To improve interoperability, the platform could provide better tools and clearer documentation for integrating with legacy systems, making the transition easier for organizations with existing infrastructure. A defined protocol or process for integration, supported by various checks and an environment for incremental testing, could be especially useful. This would allow users to test parts of the system without needing to integrate the entire MAS at once. Expanding the range of supported communication protocols between the orchestration platform and agents should not be a challenge, since the platform was originally designed with extensibility in mind. While the platform is currently best suited for simpler environments, future integration with cloud ecosystems could improve scalability, interoperability, and security, making it a stronger fit for more complex and demanding scenarios.

Below are reflections from the evaluators:

- "Yes, provided the intelligent agent exposes its functionality via a supported interface (e.g., HTTP API). APi would wrap it like any other agent." from Richard Abrich

- "Yes. Having a containerised-first approach is beneficial, as the agents and the containers may easily be distributed throughout the system network." from Bogdan Okreša Đurić

7.5.2.2 Modularity and reusability

The orchestration platform is viewed as modular and reusable, with evaluators recognizing its potential to be applied across different types of systems. Its flexibility is especially valued in environments that require coordination of distributed services, such as research setups, hybrid AI architectures, and cloud-based applications. Some evaluators, however,

noted that the platform may feel overly specialized in certain contexts, particularly where agents already share a communication protocol or where developers have limited control over how agents are started or managed. In such cases, the modularity may not provide a clear benefit. There is also an acknowledgment that, while the platform is adaptable, it may require a significant amount of manual setup and domain knowledge to use effectively in more demanding environments, such as production systems.

To improve modularity and reusability, the platform could offer automated configuration tools and reusable templates to reduce setup effort. Adding support for integrating with agents that are already running and not directly managed by the user would make the platform more adaptable. This flexibility would allow it to fit into a wider range of real-world systems, especially those with limited control over agent life cycle or external dependencies. Although connecting to an already running agent is technically feasible, the difference in life cycle visibility between managed and unmanaged agents would require significant architectural changes to ensure consistent and reliable support.

Key reflections from evaluators are below:

- "Integration is protocol-agnostic and can support API-driven agents like LLMs, as seen in the B.A.R.I.C.A. case. Wrappers can encapsulate any logic, and orchestration remains unchanged as long as the communication contract is met." from Igor Tomičić
- "I've not seen any indication that these types of agents require special handling. Apart from generally higher latency and possibly specialized hardware requirements, these function similarly to normal networked services." from Rio Kierkels

7.5.2.3 Usability

Feedback on the usability of the orchestration platform is mixed, with evaluators recognizing useful features but also pointing out areas for improvement. While the current functionality is appreciated, some evaluators suggest enhancements such as advanced monitoring, better debugging tools, and dynamic agent registration. Some evaluators also expressed interest in web-based orchestration visualization, centralized logging, and dynamic execution graphs, noting similarities with platforms like Kubernetes in terms of service discovery. There is also a call for better error handling, interface validation, and the option to switch between visual and text-based editing modes. Overall, while the platform meets the required usability needs, there is clear demand for a more intuitive and interactive user experience.

The platform could integrate stronger monitoring and debugging capabilities to help users manage orchestration flows more effectively, which is especially important in production-ready environments. Since the artifact's implementation follows common system design patterns, integrating widely used debugging tools should be feasible. Adding dynamic

agent registration and runtime validation would increase flexibility, particularly in real-time systems. Support for secure communication protocols and centralized logging could further improve data handling and overall system security. Lastly, offering a hosted cloud version could simplify onboarding for users who prefer not to manage container environments manually, a task that often requires additional effort, especially when ensuring compliance with the operational requirements needed for smooth and secure deployment.

Following are reflections from evaluators:

- "Some debugging features would be also beneficial e.g. log files about interaction among agents in real time." from Neven Vrček
- "Yes, docker is the perfect abstraction for all my use cases. The ability to run arbitrary code (as a docker container) and having the ability to define inputs and outputs it all it needs to run any business logic and agent code." from Matthias Bayer

7.5.2.4 Performance efficiency

Sentiment toward the orchestration platform's performance efficiency is mixed. Evaluators report a wide range of experiences, depending on the number of agents in their MASs, from just a few to several hundred. Concerns are raised about scalability and latency, with some users noting that the platform introduces latency due to indirect communication and lacks mechanisms like backpressure, flow control, and resource usage monitoring. Others, however, do not observe major performance drawbacks and appreciate the decoupled nature of the interface definitions, which allow for potential performance improvements without needing a full system redesign. Still, there are concerns that the central orchestration component could become a bottleneck or a single point of failure, particularly in large scale scenarios. While most evaluators agree that the platform provides a clear structure for managing distributed agents, some point out that the lack of a discovery service and centralized registry could limit its use in mission critical or geographically distributed deployments.

To improve performance efficiency, one possible strategy is to allow agents to communicate with one another directly, rather than always routing messages through the orchestration platform. This would require agents to be designed with more flexibility so they can integrate and exchange information independently when needed. Currently, the orchestration platform acts as a single unit, and if it fails, agent communication comes to a halt. A potential improvement would be to support multiple instances of the orchestration platform that share the state of agents and the overall orchestration. In such a setup, if one instance fails, agents could switch to another available instance. This approach would not only improve system resilience but also require enhancements to the platform's agent discovery capabilities to support seamless failover and coordination.

Evaluators shared the following observations:

- "Small number of latency-optimized agents. Latency optimization might be tricky in this framework, likely also scalability to lots of agents." from Peter Skvarenina
- "In case with large number of agents, the orchestrating platform is a bottleneck of the architecture and can be saturated leading to slower work of MAS. It is also a single point of failure of the architecture." from Neven Vrček

7.5.2.5 Functional Suitability

The functional suitability of the orchestration platform is viewed with a mix of appreciation and constructive criticism. While some evaluators find the platform sufficient for the majority of use cases, others highlight the need for features that better support complex and evolving environments. In particular, there is interest in enabling runtime agent registration and more dynamic execution flows, which would increase adaptability. Error handling is seen as an area for improvement, with the current support covering only basic scenarios and lacking advanced recovery mechanisms and health checks. Although the platform is capable of integrating intelligent agents, evaluators point out the limited support for streaming agents and the challenges involved in building and managing such agents effectively.

To strengthen functional suitability, the platform could benefit from introducing runtime agent registration and dynamic execution capabilities that respond to changing system states. This would require additional tracking of agent life cycle events, and potentially extending monitoring to include other aspects such as message passing. Improving system resilience could also involve adding stronger fail safe mechanisms, better validation, and periodic health checks, where each entity reports its status and bandwidth usage to the orchestration platform at regular intervals.

Following are reflections from reviewers:

- "Error handling mechanism seems basic, lacking recovery strategies. No health check are mentioned." from Peter Skvarenina
- "Indeed, it would seem that the artefact applies to integrating GenAI agents into a multiagent system. Moreso, it seems that the artefact could be used to facilitate communication between non-GenAI agents and those that do use modern intelligent solutions." from Bogdan Okreša Đurić

7.5.3 Agent definition

For the agent definition component, the primary characteristics analyzed from the artifact quality dimension include interoperability, modularity and reusability, usability, and

functional suitability. In addition, the evaluation incorporates the standard programming language criteria, focusing on readability, writability, and reliability. This combination provides a balanced view of both the structural qualities of the agent definition and the expressiveness, clarity, and robustness of the language used to define it.

7.5.3.1 Interoperability

Sentiment from evaluators regarding the interoperability of the agent definition is positive, with a diverse range of supported communication protocols being utilized. HTTP is identified as the most frequently used protocol among agents, reflecting its reliability and widespread adoption in integration scenarios. WS and Netcat are also commonly used. In addition to these, some evaluators mention protocols like XMPP, MQTT, gRPC, and Server-Sent Events (SSE), suggesting that while the current protocol support is sufficient for many use cases, there is growing interest in expanding interoperability to more specialized environments. These suggestions reflect a need to accommodate use cases such as real-time communication, IoT integration, and niche protocols used in specific domains.

Extending the set of supported protocols within the agent definition would not require significant changes. The agent definition and its corresponding adapter on the orchestration platform are designed to support the easy addition of new communication protocols, whether synchronous or asynchronous. Gen AI agents, for example, are increasingly making use of SSEs to stream information progressively. Since SSE is conceptually similar to WS, which is already supported by the platform, introducing support for SSE should be straightforward. These improvements would enhance protocol flexibility and further strengthen the platform's interoperability.

Below are key observations from evaluators:

- "XMPP/MQTT/gRPC/MCP etc. XMPP is used by my systems to allow interaction between agents and humans real-time, MQTT is used by my agents to communicate with IoT sensors, and gRPC is used by my agents to call programs on different computers." from Peter Skvarenina

- "Usually, the agents communicate using a simple API or web sockets. Sometimes, agents are in an environment where it is sufficient to use simple XMPP." from Bogdan Okreša Đurić

7.5.3.2 Modularity and reusability

Opinions on the agent definition component's modularity and reusability are divided, with clear recognition of its potential alongside suggestions for refinement. Many respondents find that the effort required to adapt an existing agent for use within the system is low to moderate, especially when the agent already supports a compatible protocol and can be

started by the platform. Some evaluators point out limitations, such as the restriction to a single input and output communication interface per agent, and the lack of support for direct output writing in request and response-based protocols. When it comes to updating agent logic after integration, most agree that internal changes do not require modifications to the communication flow specification unless the external interface is altered. The separation between communication flow and agent implementation is generally seen as a strength of the design.

Currently, the agent definition uses a single format to specify both the communication protocol and the contract with the agent. This unified approach is convenient and promotes consistency across different use cases, but it can also limit flexibility for protocol specific implementations, sometimes requiring changes to the agent's business logic or communication contract. One possible improvement would be to support separate protocol definitions tailored to each protocol, which could reduce the effort needed to adapt agent logic, though at the cost of reduced readability and maintainability. Additionally, adding support for multiple input and output communication interfaces per agent could improve flexibility but would require rethinking the structure of the agent definition and the way communication flows are specified in the orchestration layer. In particular, it would be necessary to define how the orchestration platform determines which communication interface to use when interacting with the agent.

Following are reflections from the evaluators:

- "Low effort if the agent already uses a supported protocol (HTTP, WS, STDIN, etc.), just need to write the .ad file. High effort if it uses an unsupported protocol, requires modifying the agent or creating a proxy." from Richard Abrich

- "Low to moderate effort. As long as the agent supports one of the accepted I/O formats and can be started by the platform (UNIX, Docker, Kubernetes), inclusion mostly involves defining its .ad file. No codebase changes are typically required." from Igor Tomičić

- "It seems like one constraint is that an agent can only have one input and output channel. Also the output seems to be polled by the platform. That's something to keep an eye on. Would be great to have the ability to write the outputs directly somewhere from the agent." from Matthias Bayer

7.5.3.3 Usability

While the agent definition was generally found usable by evaluators, several noted that it may pose challenges for non-technical users. Many evaluators find the process of writing and configuring agent definitions to be straightforward, particularly for those familiar with YAML. However, the format may feel repetitive when specifying basic functionality,

and some evaluators note that the current structure could benefit from simplifications or higher-level abstractions. While the agent definition is generally intuitive for developers, non-technical stakeholders may find it difficult to understand due to its technical nature. This points to a need for clearer documentation and possibly the introduction of a more user-friendly interface. Suggestions include the development of a visual or form-based editor that would allow users to configure agents through guided inputs rather than raw configuration files.

To improve usability, the system could offer tools that simplify repetitive tasks and abstract common configuration patterns, reducing manual effort. Providing a visual editor or step-by-step form that generates valid agent definitions would make the system more accessible to non-technical users. In addition, improving documentation with real-world examples and clear explanations of each supported property would help users of all backgrounds work more confidently with the agent definition. These improvements would enhance the overall usability of the component and make it more approachable for a wider audience.

Below are key observations from the reviewers:

- "I expect non-technical stakeholders will be confused by the many technical terms in the agent definitions, such as data-type, type: unix, and values such as <!eof!> probably look scary to them" from Michael van Elk
- "Since agents are defined using the parametrised approach, it is easy to modify and customise agents for their inclusion in another system. Such adaptability is most welcome." from Bogdan Okreša Đurić

7.5.3.4 Functional suitability

Evaluators' feedback on the functional suitability of the agent definition is generally positive, with most finding the supported communication protocols adequate for their needs. The current approach allows flexibility in protocol specification and is seen as an interesting and user friendly way to describe agent communication interfaces. However, some evaluators point out the need for additional features to improve operational flexibility. In particular, the lack of support for dynamic updates to agent definitions and the limited error handling capabilities are identified as areas for improvement. Nonetheless, most respondents consider the existing feature set sufficient for many real-world applications.

In the current state, any updates to an agent's communication interface require corresponding changes to the agent definition. This manual process can introduce failures if not done correctly and may lead to significant issues, especially if the agent is already running in a production environment. One possible solution to this challenge would be to allow the orchestration platform to retrieve the agent definition directly from the agent itself, rather than relying on a manually written configuration file. For example, each

agent could expose a common interface that the orchestration platform queries at startup to fetch the communication contract. This contract could then be used to automatically configure the appropriate communication adapter, reducing the risk of mismatches and simplifying the update process.

Key reflections from the evaluators are as follows:

- "Would love to see error handling, auto-scaling and maybe a health check to see if the agent properly started up" from Matthias Bayer
- "APi does not support dynamic changes to agent definitions which might suppress its suitability for fast changing environments." from Peter Skvarenina

7.5.3.5 Readability

The readability characteristics of the agent definition is viewed positively. Most evaluators find the syntax clear and easy to work with, especially those familiar with YAML and tools like Docker. The YAML-like structure is appreciated for its simplicity, allowing developers to define agents in a straightforward manner. Some concerns were raised about repetitive patterns in basic configurations and the ambiguity of certain values, which may not be clearly distinguishable as special syntax or plain strings. While developers find the format intuitive, non-technical stakeholders may struggle to understand the definitions without supporting explanations or documentation. Technical terms and protocol references can be confusing to those without a technical background, highlighting a gap in accessibility.

Introducing templates in YAML could help reduce boilerplate and make the process more straightforward by minimizing repeated code, though this might come at the cost of reduced clarity in some cases. For non-technical stakeholders, developing a visual or form-based editor could significantly improve accessibility, allowing them to interact with agent definitions without requiring deep technical knowledge. Certain properties within the agent definition, such as message format, delimiter, and related configuration details, may still require a basic level of technical understanding to configure correctly.

Some of the reflections from the evaluators are listed below:

- "Developers familiar with Docker or Compose will find the structure intuitive. Templates could help reduce boilerplate further." from Igor Tomičić
- "...(agent definition) is very easy to read also thanks to the very common vocabulary used." from Matthias Bayer

7.5.3.6 Writability

Evaluators' feedback on the writability of the agent definition is largely positive, particularly in terms of how clearly the configuration allows developers to express agent behavior

and protocol interactions. The structure is considered flexible enough to capture a wide range of communication protocols, and the available properties are viewed as sufficiently comprehensive for most integration scenarios. Several evaluators note that the format supports a logical and consistent way of defining agents, which contributes to a smooth authoring experience.

Providing inline suggestions or lightweight editing tools that assist during authoring could preserve flexibility while making the process of writing agent definitions faster and more efficient. Clarifying the purpose and usage of each property in the supporting material would also help users better leverage the expressiveness of the format, particularly in more advanced protocol configurations. Additionally, offering an interface that accepts natural language input could further improve writability, especially for nontechnical stakeholders who may find the syntax unfamiliar.

Key reflections on the observed characteristics are as follows:

- "Yes, the simplicity and YAML-like syntax make it easy to write." from Michael van Elk
- "Although quite some technical knowledge is required to configure it directly. For example, non-techinical people will probably not know why or when to use the HTTP protocol vs raw TCP, what a delimiter is in the context of message parsing or what JSON is." from Rio Kierkels

7.5.3.7 Reliability

Evaluators generally perceive the reliability of the agent definition component positively, primarily because it relies on well known standards and does not introduce unfamiliar elements or concepts. Some note that adapting request and response-based protocols into an input and output-based format can create opportunities for misuse, which may lead to failures if not handled correctly. This approach may also negatively affect characteristics such as latency, as it prevents certain protocols from leveraging their intrinsic features to their full potential.

Given the limited set of supported properties in the agent definition, which are designed to be generic enough to support a wide variety of protocols, not all protocol-specific functionalities can be fully utilized. This may affect latency or influence the overall behavior of agents in certain situations. Similarly, some fail-safe mechanisms might not be used to their full extent. To improve reliability, it would be beneficial to strengthen validation of the agent definition and introduce more robust guardrails within the orchestration platform, tailored to the characteristics and requirements of each supported protocol.

Some of the evaluator reflections are as follows:

- "Latency optimization often requires dynamically changing agents on the fly, reacting to their state and state of other agents" from Peter Skvarenina

- "...however after the initial conversion (of agent into the orchestration platform), further updates likely would not require further changes." from Michael van Elk

7.6 Objectives fulfillment

In Chapter 4, the high-level objectives of the artifact are laid out, defining several critical requirements the artifact should fulfill. These include cloud support, agent orchestration, and similar capabilities. Below, each individual component is analyzed along with evaluators' feedback regarding its fulfillment:

- AI support: Since the implementation of agents is left to the engineer, and both the orchestration platform and programming language are agnostic to the agent internals, agents can implement any AI-based business logic. Evaluators confirmed this flexibility and reported no concerns regarding the integration of intelligent business logic.
- Flexibility and control of communication flow: The programming language is based on the π-Calculus, which is tailored for modeling communication in concurrent systems [76]. Within the agents' orchestration specification, engineers can define communication flows between agents and other entities by specifying direction, participants, protocol, and transformations from input to output. Evaluators noted that the language is largely intuitive and allows for detailed and effective communication flow definitions.
- Distributed agents orchestration: The orchestration platform, responsible for coordinating and synchronizing agents, has been developed with support for distributed deployments and cloud environments in mind. Execution flows allow specifying startup order (e.g., sequential, parallel), offering flexible orchestration options. Evaluators expressed satisfaction with the artifact's current support for distributed agents.
- Specification and control of agent execution order: As previously mentioned, execution order can be defined using a variety of operators, enabling both basic and more specialized use cases. Most evaluators agreed that the existing operators are sufficient, although some suggested that it would be valuable to support agent startup triggered by message-based events, not only life cycle events.
- Support for agent hierarchies: Agent hierarchies are supported through holonic systems and the introduction of environment interfaces, which facilitate communication between holons. The environment concept enhances communication flexibility. Evaluators accepted the concept and its implementation, though noted it is not something they frequently use in practice.

- Ease of integration into existing systems: The artifact is designed for straightforward integration into existing systems. This was confirmed by evaluators, who noted that the integration effort is minimal. The orchestration platform is agnostic to the agent implementation, allowing engineers to use any preferred technology. Predefined communication protocols cover most use cases, though more specialized scenarios may require additional protocols.

- Support for agents using different technologies and protocols: Engineers are free to implement agents using any technology, as long as the communication interfaces adhere to supported protocols. Most evaluators found the current set of protocols sufficient, though they acknowledged that future use cases may demand broader protocol support.

- Ease of component replacement: Adding, removing, or updating agents and communication flows is easily accomplished by updating the orchestration specification or modifying agent definitions to reflect changes in communication contracts. Frequent changes are considered in the implementation. Evaluators agreed that component replacement is straightforward and incurs minimal effort.

- Operational scalability: The orchestration platform manages communication and agent startup, but does not itself handle significant processing workloads. This makes it suitable for scaling, assuming adequate underlying resources. However, the platform's centralized architecture may become a bottleneck if a critical error occurs. Evaluators noted that their systems typically include only a small number of agents, and thus scalability concerns are minimal in such contexts.

- Resilience and fault tolerance: The platform monitors agent life cycles, enabling it to follow execution flows and perform recovery actions such as restarting agents when needed. This contributes to its fault tolerance. Evaluators acknowledged this capability but suggested that improved type safety in the programming language could make the system more predictable and robust.

- Support for cloud computing: The orchestration platform allows agents to be deployed as UNIX processes, Docker containers, or Kubernetes instances across distributed environments. However, it currently lacks built-in support for configuring agent networks in distributed cloud environments, which must be handled externally. Evaluators considered the existing cloud support adequate, but noted that making the orchestration platform cloud-native would be a valuable enhancement.

Based on the analysis of the predefined requirements, the implemented artifact, and feedback from evaluators, it can be concluded that all high-level requirements for the artifact have been met. Some requirements have been addressed more comprehensively

than others, but any remaining areas for improvement can be developed further in future iterations.

Chapter 8

Discussion

The integration of the implemented artifact into O HAI 4 Games use cases, along with evaluations from project team members and independent experts, has highlighted both the strengths of the artifact and areas where further improvement is possible. Opportunities for enhancement are particularly in aspects such as error handling mechanisms, support for additional communication protocols, and related technical considerations.

One of the areas highlighted for improvement is the artifact's error handling and fail-safe capabilities. This includes more rigorous type checking within the programming language used to describe both the agents' orchestration specifications and the agent definitions themselves. Although the orchestration platform currently supports basic error handling mechanisms, particularly in managing agent life cycle events, there is room to enhance these safeguards further. For example, additional mechanisms could be introduced to handle failures related to the orchestration platform itself, communication protocol disruptions between agents, and unexpected runtime conditions. Strengthening these aspects would contribute to a more resilient and dependable system, especially in large-scale or mission-critical deployments where reliability is essential.

The programming language and its elements were widely accepted by evaluators, largely due to their intuitiveness. Nonetheless, some evaluators noted that the concept of a holon is less familiar, and suggested that its naming and possibly its implementation be reconsidered. In the current artifact, holons and their environments share many similarities with the channel entity type, leading some to propose unifying the two. While these suggestions are reasonable, maintaining a clear distinction between the two and allowing users to incorporate holonic features on demand provides greater flexibility and supports more advanced hierarchical modeling when needed, without imposing additional complexity on simpler use cases.

On the subject of holons, they can communicate with one another only if they are started through the orchestration platform and specifically use input and output environment interfaces. An agent started through the orchestration platform can be accessed by external agents, including those not started as part of the orchestration platform. How-

ever, this flexibility does not extend to holon environments, which limits how holons can interact across boundaries. Adding more flexibility in this area, and specifically allowing for external communication, would be highly beneficial. One possible approach is to introduce additional configuration options for holon environments, similar to how agent definitions are handled. This would allow users to specify the desired communication interface for a holon, while the orchestration platform manages the inbound and outbound communication accordingly.

Artifact evaluators noted that the agents' orchestration specification is easy to read, thanks to its simple primitives, intuitive structure, and clear syntax. However, in systems with a large number of agents, the specification can become cluttered and difficult to follow. To address this, several evaluators suggested developing a visualization tool that can parse the orchestration specification and display agent communication flows in a more digestible format. This could be achieved either by building a custom visualization interface or by creating a conversion tool that translates the specification into an existing visualization language, such as Mermaid.

One of the evaluators also suggested introducing more parametrization and template support into the programming language. Parametrization is currently used for agents, which helps reduce verbosity in the agents' orchestration specification, as a single line can replace at least two lines, especially for agents with multiple communication flows. Expanding parametrization to other concepts, such as channel or environment message transformations, could further simplify the specification. Additionally, it is likely that multiple holons, defined in separate specification files, would share common declarations such as channels or environments. By introducing templates that can be stored in a shared file, users could import them into their individual holon specifications, promoting reuse and consistency across different parts of the system.

Less technical users might struggle to write the agents' orchestration specification or agent definition, as there is at least some learning curve involved. One proposal to address this is the use of NLP middleware, such as LLMs, which could take natural language input from the user and convert it into a valid specification. To support this, the model would need to understand the semantics, pragmatics, and structural elements of the language, which may not pose a significant challenge given the language's simplicity. In addition, the output from the model could be passed through a validator to check for correctness. An alternative to the natural language interface would be a command-line tool that generates templates. This generator could prompt the user with a series of guided questions such as "How many agents are there?" or "Which channels or environments does agent X communicate with?". Based on the user's answers, a valid specification could be produced through a more deterministic and structured approach.

The orchestration platform currently operates on a static, predefined agents' orchestration specification, meaning that all agents, their communication flows, and execution

flows are known in advance. At this stage, there is no flexibility to run agents dynamically or on demand. Introducing such a feature could be valuable for fast-paced and adaptive environments, where responsiveness and scalability are critical. It could also support better load balancing by allowing multiple instances of a specific agent to be started based on the number of active users. However, while this addition would improve adaptability, it may also reduce transparency, making it more difficult to maintain a clear understanding of which agents are active within the MAS at any given time.

The orchestration platform in its current form is implemented in a way that only allows it to start agents that are explicitly included in the orchestration. This is because having visibility into an agent's life cycle is crucial for knowing when the agent starts, becomes ready, or is shutting down. Evaluators noted that the ability to integrate running or active agents into the orchestration would be highly beneficial, especially as users begin to develop more publicly accessible agents. While this improvement is certainly feasible, it may require significant changes to both the orchestration platform and the agent wrapper, particularly regarding how failures are handled. If such an agent goes down, the orchestration platform would not be able to restart it. As a result, it may be necessary to restrict supported execution flows for these agents, so that other agents' execution cannot rely on them unless they were started through the orchestration platform.

Most evaluators stated that the current set of execution flows and corresponding operators is sufficient for the majority of use cases they encounter. However, some raised interesting suggestions around enabling execution flows that are not solely tied to the agent life cycle but are also reactive to messages. For example, if a large volume of messages targets a particular agent or channel, it could be useful to automatically spawn new agent or channel instances in response to traffic load. Similarly, allowing an agent to trigger the startup of another agent could enable the design of an orchestrator agent. This kind of improvement could be achieved by allowing agents not only to send messages but also to request specific actions from the orchestration platform. To support this, agents would need to implement additional capabilities, which would make them more tightly coupled to the orchestration platform. This, however, conflicts with the requirement that agents should remain as agnostic to the orchestration platform as possible. In a related direction, since the orchestration platform has visibility into message traffic, a monitor-type entity could be introduced at the programming language level, defined by a set of rules and corresponding actions to be triggered when conditions are met.

Evaluators also suggested expanding the number of supported protocols for agent communication. While the currently supported protocols cover most common use cases, there are niche scenarios that require more specialized ones. Adding support for additional protocols should be relatively straightforward, although the complexity depends on how the protocol operates, particularly whether it is asynchronous or synchronous. The agent definition currently separates the communication interface for input and output, which

is useful in situations where the agent uses different protocols for receiving and sending messages or when dealing with asynchronous communication. However, the current implementation relies on a generic format that aims to be flexible enough for most protocols. As a consequence, if an agent communicates using the same protocol for both input and output, this cannot be expressed as a single interface. This limitation can lead to the loss of protocol-specific features and may introduce additional latency or an increased risk of failure in systems that are sensitive to performance. This challenge could be overcome by adjusting the agent definition format and aligning it more appropriately with the characteristics of each individual protocol.

The artifact is open-source, and anyone is welcome to contribute to its further development. One area of the orchestration platform that may require more changes to support user-specific use cases is the agent definition, particularly in relation to adding support for additional protocols. While adding new protocols is relatively straightforward, it still requires understanding the internal workings of the platform in order to modify the appropriate parts of the codebase. To address this, decoupling the logic for agent communication protocol extensions from the core of the orchestration platform could be beneficial. This would make the platform more modular and easier to navigate, allowing users to focus only on the communication protocol and its implementation when making adjustments.

In the current setup, the orchestration platform acts as a centralized unit responsible for starting agents, tracking their state and life cycle, and managing message passing between agents through channels or environments. This design was chosen to provide better visibility and control over agents. However, it comes at the cost of being a single point of failure. Some evaluators suggested adding an alternative orchestration strategy in which the platform remains responsible for starting agents, but once active, agents can communicate directly with each other. In such a setup, even if the orchestration platform fails, agents would still be able to continue interacting. Nonetheless, if the platform goes down, no new agents can be started, and execution flows can no longer be triggered or coordinated. Another proposed alternative involves deploying multiple orchestration platform instances that share the same state, so if one instance fails, others can continue facilitating communication between agents and other entities. Building on this idea, a load balancing strategy could be introduced where, in larger systems, multiple orchestration platform instances each manage a subset of agents. This would improve reliability and also enhance scalability since no single platform instance would be responsible for the entire system.

As mentioned previously, agents can only communicate with one another through the orchestration platform. This design shifts the burden of communication away from the agent implementation and onto the platform itself, allowing agents to remain flexible regarding who they communicate with. However, this approach comes at the cost of per-

formance, particularly in terms of latency, since all communication must pass through an intermediary. In scenarios where performance is critical and latency must be kept as low as possible, this may pose a limitation. To address this, it may be useful to introduce multiple communication strategies, allowing users to choose whether agents should communicate through the orchestration platform, with minimal implementation effort and access to features like channel broadcasting, or communicate directly, which would require more logic in the agent implementation and would exclude certain orchestration specific capabilities.

High standards for agent authentication and overall security are not comprehensively implemented in the orchestration platform. Although some basic logic has been implemented and can be easily extended, it is not enforced in any meaningful way. To ensure the artifact can be used in production environments, more attention needs to be given to security and authentication. The primary responsibility could lie with the orchestration platform, which would verify all incoming and outgoing messages between agents and other entities. However, some safeguards may also need to be implemented on the agent side to ensure that agents accept messages only from known senders, such as the orchestration platform itself.

The orchestration platform does not currently offer verbose logging or advanced observability features. This limitation makes it more challenging to operate in a production environment, as users lack clear visibility into the platform's behavior, including which messages are being exchanged between agents, the life cycle status of agents, and overall resource utilization. Introducing a graphical user interface similar to those used in Docker or Kubernetes could significantly improve monitoring and make it easier to track and manage the orchestration platform in real-time.

The artifact and its implementation have been documented to a good extent. Nonetheless, adding more examples that cover various aspects such as agents' orchestration specifications, agent definitions, and corresponding implementations would be valuable. This would give users concrete examples that align with their own use cases, helping to reduce the learning curve and improve familiarity with the platform. Additionally, domain-specific tutorials and targeted use case guides could further support adoption and ease of use.

Overall, the discussion reveals how design decisions shaped the artifact's current capabilities and limitations. While many of the identified challenges are technical in nature, they also reflect deeper trade-offs between control, flexibility, and ease of use. These reflections not only inform future iterations of the platform but also contribute to a broader understanding of what is required to design agent-oriented systems that are both expressive and operational in real-world contexts.

Chapter 9

Conclusion

The objectives of this research focused on the development of a programming language and an associated declarative engine that enable the orchestration of heterogeneous microservices within a MAS architecture, while also supporting AI, cloud-based environments, and holonic systems [31]. Following an in-depth analysis of industry practices through comparison of existing tools specialized in agent development and orchestration, as well as a review of scientific literature in the relevant domains and the requirements of the O HAI 4 Games project, opportunities for improvement and contribution were identified. Scientific literature primarily focuses on the application and protocol layers, offering limited insight into coordination mechanisms [24, 33, 3, 17]. Industry practices emphasize infrastructure and application layers, providing practical tools but lacking explicit support for communication flow specification [110, 18]. Both reveal gaps that leave room for improvement in orchestrating distributed systems more effectively. The literature analysis was also used to define a set of requirements that guided the creation of a new artifact. This research was conducted using the Design Science Research (DSR) paradigm, which is particularly suitable given that the goal of the research was to develop a novel artifact [41].

In this research, agents orchestration is viewed through the lens of communication flow specification. This refers to defining which agents and related entities communicate with one another and what the communication configuration is. Orchestration also includes the order in which agents are started.

The high-level requirements for the artifact included support for intelligent agents, flexible specification and control of communication flows, support for orchestrating distributed agents, the ability to define agent execution order, support for agent hierarchies through holonic systems, ease of integration with existing systems, compatibility with agents developed using different technologies and protocols, ease of component replacement, operational scalability, resilience and fault tolerance, and support for cloud environments.

The resulting artifact, named `APi`, consists of three core components: the agents or-

chestration specification, the orchestration platform, and the agent definition. The orchestration specification defines which agents communicate and is described using the newly developed programming language. This language is based on π-Calculus [76], chosen for its suitability in describing communication flows in concurrent systems. The language was developed using the ANTLR framework [75]. The second component, the Python-based orchestration platform, includes the declarative engine and is responsible for consuming the orchestration specification, starting agents in the correct order, and enabling coordination, communication, and synchronization among them. The third component, the agent definition, describes how each agent is started and how the orchestration platform communicates with it. Since one of the artifact requirements was to support business logic implemented in various technologies, the orchestration platform must be aware of how to access and communicate with each agent.

The artifact ecosystem includes five elements: agent, channel, environment, holon, and execution flow. An agent refers to a service developed by an engineer that collaborates with other agents to achieve a shared goal. Agents never communicate directly but instead interact through channels and environments. Channels are responsible for transferring messages. An agent may send a message to a channel or receive one from it. A holon is a semantic grouping of agents. The platform supports the collaboration of multiple holons, which interact through an environment interface. Each holon may have an input and an output environment. Agents that need to receive output from other holons subscribe to messages arriving through the input environment, while those sharing output send messages to the output environment. Execution flow refers to the order in which agents are started. Supported flows include sequential, parallel, restart, and conditional execution based on agent completion.

The integration of the artifact was demonstrated through four use cases developed as part of the O HAI 4 Games project [64]. These include MMORPGs, cognitive agents and gamification, autonomous vehicles, and a game streaming system. In some cases, the artifact was integrated into existing systems. In other cases, systems were developed with the artifact in mind from the beginning. This demonstrates the artifact's adaptability. These use cases were also used to evaluate integration effectiveness and to identify and resolve issues during implementation. They provided insight into both strengths and limitations of the artifact.

The artifact was evaluated by members of the O HAI 4 Games team and independent experts with experience in building AI and agent-oriented systems. Evaluation methods included interviews, a questionnaire, and ongoing feedback such as bug reports, feature requests, and observations. Each artifact component was assessed along two dimensions: artifact quality and standard programming language criteria. Artifact quality characteristics were derived from the `ISO/IEC 25002:2024` standard and measured in terms of interoperability, modularity and reusability, usability, performance efficiency, and functional

suitability. Standard programming language criteria included readability, writability, and reliability [114]. Thematic analysis was used to analyze the collected feedback [62].

The results indicated a broadly positive sentiment across all components. The programming language for specifying communication flows was appreciated for its simplicity and minimalistic design, while still supporting key objectives such as a holistic view of multi-agent configurations and flexible communication. The orchestration platform was valued for its ability to manage agents based on the specification. Features such as support for cloud environments, intelligent agents, and communication proxies were highlighted as especially useful. The agent definition component was praised for enabling flexible specification of communication protocols between agents and the orchestration platform. Based on the evaluation, the artifact requirements were successfully fulfilled.

Identified areas for improvement include support for dynamic agent allocation at runtime, instead of requiring predefined agents and flows. There were also requests for the ability to connect with existing agents not started by the platform, which will be valuable in scenarios where agents are publicly accessible. Observability of orchestration and execution was another mentioned request. Adding such features could improve flexibility and system reliability.

The answer to the research question "What types of communication flows should be supported by the programming language considering the needs of modern domains related to microservice architecture, artificial intelligence, and cloud computing?" emphasizes the importance of supporting structured and flexible communication among distributed components. As detailed in chapter 4, the developed language enables the definition of communication between agents, channels, environments, and holons, while also capturing the configuration of such communication. This includes specification of the communication protocol, the direction of communication, and the transformation of messages as they move through the system.

In response to the research question "How to shape the syntax, semantics, and pragmatics of a programming language for the orchestration of heterogeneous microservices in multi agent system architectures by utilizing process calculus?" the language was designed to be simple to use while still supporting key ideas from process calculus and more specifically π-Calculus [76]. Chapters 4 and 5 present the implementation details. Programming language syntax is easy to read and write, helping developers clearly describe how agents interact. The semantics are based on π-Calculus principles, allowing agents to collaborate through different execution flows such as sequential, parallel, conditional, and restart. The goal was to build a language that makes it easier to define how agents work together, while keeping the overall orchestration clear and manageable.

Regarding the question "How to support the design process of complex methods ensemble utilizing holonic systems?", the artifact implements support for holonic structures by introducing input and output environments. Chapters 4 and 5 outline how these en-

vironments function as communication interfaces between holons and other system components, promoting modular design, clear separation of concerns, and collaboration across nested agent groups.

Given the favorable evaluation results across all assessed dimensions and the successful fulfillment of the artifact requirements detailed in chapters 4 and 7, it can be concluded that the research objectives "Develop a programming language that enables the orchestration of heterogeneous microservices in the multi agent systems architecture, artificial intelligence, and cloud computing" and "Develop a declarative engine based on process calculus capable of controlling communication flows between intelligent agents" have been achieved.

Considering the demonstrated improvements in agent orchestration and communication flows specification enabled by the developed programming language, and supported by the feedback presented in chapter 7, the hypothesis "Programming language for communication flows specification based on process calculus shall enhance the orchestration of heterogeneous microservices using multi agent systems" is confirmed by the outcomes and contributions of this research.

Future research will address the identified limitations and continue evolving the artifact to align with ongoing advances in Gen AI and architectures based on MASs. The focus will be on expanding the artifact's capabilities, increasing its adaptability to diverse use cases, and ensuring its relevance in increasingly complex and intelligent distributed environments.

Bibliography

[1] E. Adam, R. Mandiau and C. Kolski. 'Homascow: a holonic multi-agent system for cooperative work'. In: *Proceedings 11th International Workshop on Database and Expert Systems Applications*. IEEE. 2000, pp. 247–253.

[2] AILab-FOI. *Python-Glulxe: A Glulx interpreter in Python*. GitHub repository. 2025.

[3] A. Althnian and A. Agah. 'Evolutionary learning of goal-oriented communication strategies in multi-agent systems'. In: *Journal of Automation Mobile Robotics and Intelligent Systems* 9.3 (2015), pp. 52–64.

[4] N. Antonopoulos and L. Gillam. *Cloud computing*. Vol. 51. 7. Springer, 2010.

[5] F. o. O. Artificial Intelligence Laboratory and U. o. Z. Informatics. *HoloGameEngine: Holographic Game Engine*. `https://github.com/AILab-FOI/HoloGameEngine`. Accessed: 2025-04-26. 2025.

[6] *AutoGen*. `https://microsoft.github.io/autogen/`.

[7] J. C. Baeten. 'A brief history of process algebra'. In: *Theoretical Computer Science* 335.2-3 (2005), pp. 131–146.

[8] J. C. Baeten and M. A. Reniers. *Process algebra: equational theories of communicating processes*. 50. Cambridge university press, 2010.

[9] M. Barbuceanu and M. S. Fox. 'The design of a coordination language for multi-agent systems'. In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1996, pp. 341–355.

[10] S. Baškarada, V. Nguyen and A. Koronios. 'Architecting microservices: Practical opportunities and challenges'. In: *Journal of Computer Information Systems* (2020).

[11] F. Bellifemine et al. 'JADE—a java agent development framework'. In: *Multi-agent programming: Languages, platforms and applications* (2005), pp. 125–147.

[12] J. A. Bergstra, A. Ponse and S. A. Smolka. *Handbook of process algebra*. Elsevier, 2001.

[13] M. Berna-Koes, I. Nourbakhsh and K. Sycara. 'Communication efficiency in multiagent systems'. In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004.* Vol. 3. IEEE. 2004, pp. 2129–2134.

[14] R. H. Bordini, J. F. Hübner and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason.* John Wiley & Sons, 2007.

[15] A. Bozkurt. 'Generative artificial intelligence (AI) powered conversational educational agents: The inevitable paradigm shift'. In: *Asian Journal of Distance Education* 18.1 (2023).

[16] E. Brynjolfsson, D. Li and L. R. Raymond. *Generative AI at work.* Tech. rep. National Bureau of Economic Research, 2023.

[17] L. Búrdalo et al. 'The information flow problem in multi-agent systems'. In: *Engineering Applications of Artificial Intelligence* 70 (2018), pp. 130–141.

[18] B. Burns et al. *Kubernetes Best Practices.* " O'Reilly Media, Inc.", 2023.

[19] B. Burns et al. *Kubernetes: up and running.* " O'Reilly Media, Inc.", 2022.

[20] E. Casalicchio and S. Iannucci. 'The state-of-the-art in container technologies: Application, orchestration and security'. In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5668.

[21] Clarivate. *Web of Science.* `https://www.webofscience.com/wos/`. Accessed: 2025-04-23. 2025.

[22] E. Cole. *Network security bible.* John Wiley & Sons, 2011.

[23] *Concepts Introduction.* Accessed: 2024-06-02.

[24] D. D. Corkill. 'Blackboard Systems'. In: 1991.

[25] M. Cullinan and D. Coulter. 'A Holonic Homoiconic Agent-Oriented Patricia Trie'. In: *Proceedings of the Future Technologies Conference.* Springer. 2023, pp. 329–343.

[26] N. Dragoni et al. 'Microservices: yesterday, today, and tomorrow'. In: *Present and ulterior software engineering* (2017), pp. 195–216.

[27] Elsevier. *Scopus Preview.* `https://www.scopus.com`. Accessed: 2025-04-23. 2025.

[28] W. Fokkink. *Introduction to process algebra.* springer science & Business Media, 2013.

[29] S. Forrest et al. 'A sense of self for unix processes'. In: *Proceedings 1996 IEEE symposium on security and privacy.* IEEE. 1996, pp. 120–128.

[30] M. R. Genesereth. 'Software Agents Michael R. Genesereth Logic Group Computer Science Department Stanford University'. In: (1994).

[31] C. Gerber, J. Siekmann and G. Vierke. 'Holonic multi-agent systems'. In: (1999).

[32] G. Giacobbi. *The GNU Netcat project*. `https://netcat.sourceforge.net/`. 2006.

[33] C. V. Goldman and S. Zilberstein. 'Optimizing information exchange in cooperative multi-agent systems'. In: *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. 2003, pp. 137–144.

[34] C. Gong et al. 'The characteristics of cloud computing'. In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 275–279.

[35] R. Goodwin. 'Formalizing properties of agents'. In: *Journal of Logic and Computation* 5.6 (1995), pp. 763–781.

[36] Google LLC. *Google Cloud Run*. `https://cloud.google.com/run?hl=en`. Accessed: 2025-04-23. 2025.

[37] Google LLC. *Google Stadia*. Accessed: 2025-05-08. 2023.

[38] D. Grune and C. J. Jacobs. *Parsing Techniques (Monographs in Computer Science)*. 2006.

[39] T. J. Hacker, B. D. Athey and B. Noble. 'The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network'. In: *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE. 2002, 10–pp.

[40] C. Hattingh. *Using Asyncio in Python: understanding Python's asynchronous programming features*. " O'Reilly Media, Inc.", 2020.

[41] A. R. Hevner et al. 'Design Science in Information Systems Research'. In: *MIS Quarterly* 28.1 (2004), pp. 75–105. ISSN: 02767783. DOI: `10.2307/25148625`. arXiv: `/dl.acm.org/citation.cfm?id=2017212.2017217 [http:]`.

[42] E. B. Hunt. *Artificial intelligence*. Academic Press, 2014.

[43] IEEE. *IEEE Xplore*. `https://ieeexplore.ieee.org/`. Accessed: 2025-04-23. 2025.

[44] S. Kagaya et al. 'An application of multi-agent simulation to traffic behavior for evacution in earthquake disaster'. In: *Journal of the Eastern Asia Society for Transportation Studies* 6 (2005), pp. 4224–4236.

[45] L. Kalita. 'Socket programming'. In: *International Journal of Computer Science and Information Technologies* 5.3 (2014), pp. 4802–4807.

[46] J. Kanclirz. *Netcat power tools*. Elsevier, 2008.

[47] J. M. Kizza, W. Kizza and Wheeler. *Guide to computer network security*. Vol. 8. Springer, 2013.

[48] T. Kubernetes. 'Kubernetes'. In: *Kubernetes. Retrieved May* 24 (2019), p. 2019.

[49] S. Kumar and U. Kumar. 'Java agent development framework'. In: *International Journal Research* 1.9 (2014), pp. 1022–1025.

[50] *LangGraph*. `https://python.langchain.com/docs/langgraph/`.

[51] W. Lepuschitz. 'Self-reconfigurable manufacturing control based on ontology-driven automation agents'. PhD thesis. Technische Universität Wien, 2018.

[52] M. Liu et al. 'Review of digital twin about concepts, technologies, and industrial applications'. In: *Journal of manufacturing systems* 58 (2021), pp. 346–361.

[53] M. Lu. *The World's Biggest Cloud Computing Service Providers*. Accessed: date-of-access. Mar. 2024. URL: `https://www.visualcapitalist.com/the-worlds-biggest-cloud-computing-service-providers/`.

[54] M. Luck. *Multi-Agent Systems and Applications: 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School, EASSS 2001, Prague, Czech Republic, July 2-13, 2001. Selected Tutorial Papers*. Vol. 2086. Springer Science & Business Media, 2001.

[55] L. Luncean and A. Becheru. 'Communication and interaction in a multi-agent system devised for transport brokering'. In: *Proceedings of the 2015 Balkan Conference on Informatics: Advances in ICT, Craiova, Romania*. 2015, pp. 2–4.

[56] N. Marathe, A. Gandhi and J. M. Shah. 'Docker swarm and kubernetes in cloud computing environment'. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE. 2019, pp. 179–184.

[57] S. MarkusMarkus et al. 'An Orchestrated Game Streaming System for Transport and Mobility Research'. In: *Transportation research procedia* 73 (2023), pp. 126–132.

[58] S. D. McArthur et al. 'Multi-agent systems for power engineering applications—Part I: Concepts, approaches, and technical challenges'. In: *IEEE Transactions on Power systems* 22.4 (2007), pp. 1743–1752.

[59] S. D. McArthur et al. 'Multi-agent systems for power engineering applications—Part II: Technologies, standards, and tools for building multi-agent systems'. In: *IEEE Transactions on power systems* 22.4 (2007), pp. 1753–1759.

[60] Microsoft Corporation. *Azure Container Instances*. `https://azure.microsoft.com/en-us/products/container-instances`. Accessed: 2025-04-23. 2025.

[61] S. Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[62] K. A. Neuendorf. 'Content analysis and thematic analysis'. In: *Advanced research methods for applied psychology*. Routledge, 2018, pp. 211–223.

[63] NVIDIA Corporation. *NVIDIA GeForce NOW*. Accessed: 2025-05-08. 2025.

[64] *OHAI4Games*. `http://dragon.foi.hr:8888/ohai4games`. Accessed: 2025-04-26.

[65] F. Ortin et al. 'An empirical evaluation of Lex/Yacc and ANTLR parser generation tools'. In: *Plos one* 17.3 (2022), e0264326.

[66] M. C. Oussalah. *Software architecture 1*. Wiley Online Library, 2023.

[67] L. Padgham and M. Winikoff. *Developing intelligent agent systems: A practical guide*. John Wiley & Sons, 2005.

[68] C.-V. Pal et al. 'A review of platforms for the development of agent systems'. In: *arXiv preprint arXiv:2007.08961* (2020).

[69] J. Palanca et al. 'A flexible agent architecture in SPADE'. In: *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer. 2022, pp. 320–331.

[70] J. Palanca et al. 'Spade 3: Supporting the new generation of multi-agent systems'. In: *IEEE Access* 8 (2020), pp. 182537–182549.

[71] J. S. Park et al. 'Generative agents: Interactive simulacra of human behavior'. In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 2023, pp. 1–22.

[72] T. Parr. 'The definitive ANTLR 4 reference'. In: (2013).

[73] T. Parr and A. Contributors. *ANTLR XML Lexer Grammar*. Accessed: 2025-03-08. 2013.

[74] T. Parr and K. Fisher. 'LL (*) the foundation of the ANTLR parser generator'. In: *ACM Sigplan Notices* 46.6 (2011), pp. 425–436.

[75] T. J. Parr and R. W. Quong. 'ANTLR: A predicated-LL (k) parser generator'. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[76] J. Parrow. 'An introduction to the π-calculus'. In: *Handbook of process algebra*. Elsevier, 2001, pp. 479–543.

[77] K. Peffers et al. 'A design science research methodology for information systems research'. In: *Journal of management information systems* 24.3 (2007), pp. 45–77.

[78] T. Peharda. *Awkward -nguin (APi): Microservice Orchestration Language*. `https://github.com/AILab-FOI/APi`. MIT License. 2025.

[79] B. C. Pierce and D. N. Turner. 'Concurrent objects in a process calculus'. In: *International Workshop on Theory and Practice of Parallel Programming*. Springer. 1994, pp. 187–215.

[80] B. C. Pierce and D. N. Turner. 'Pict: a programming language based on the Pi-Calculus.' In: *Proof, language, and interaction*. Citeseer. 2000, pp. 455–494.

[81] J. Pitt and A. Mamdani. 'Communication protocols in multi-agent systems: a development method and reference architecture'. In: *Issues in agent communication*. Springer, 2000, pp. 160–177.

[82] *Platform for Multi AI Agents Systems*. `https://www.crewai.com/`.

[83] S. Pokala. *Generative AI Trends And Use Cases To Follow In 2024*. Feb. 2024. URL: `https://www.forbes.com/sites/forbestechcouncil/2024/02/21/generative-ai-trends-and-use-cases-to-follow-in-2024/?sh=679b8d86740a`.

[84] N. Prat, I. Comyn-Wattiau and J. Akoka. 'A taxonomy of evaluation methods for information systems artifacts'. In: *Journal of Management Information Systems* 32.3 (2015), pp. 229–267.

[85] T. Rak. 'Performance Evaluation of an API Stock Exchange Web System on Cloud Docker Containers'. In: *Applied Sciences* 13.17 (2023), p. 9896.

[86] M. V. Ramos and R. J. de Queiroz. 'Formalization of simplification for context-free grammars'. In: *arXiv preprint arXiv:1509.02032* (2015).

[87] A. Reed. *Creating interactive fiction with Inform 7*. Course Technology Press, 2010.

[88] B. Ribeiro et al. 'Python-based ecosystem for agent communities simulation'. In: *International Workshop on Soft Computing Models in Industrial and Environmental Applications*. Springer. 2022, pp. 62–71.

[89] S. Rodriguez et al. 'Holonic multi-agent systems'. In: *Self-Organising Software: From Natural to Artificial Adaptation* (2011), pp. 251–279.

[90] C. K. Rudrabhatla. 'Comparison of event choreography and orchestration techniques in microservice architecture'. In: *International Journal of Advanced Computer Science and Applications* 9.8 (2018).

[91] J. Rufino et al. 'Orchestration of containerized microservices for IIoT using Docker'. In: *2017 IEEE International Conference on Industrial Technology (ICIT)*. IEEE. 2017, pp. 1532–1536.

[92] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. pearson, 2016.

[93] P. Saint-Andre. *Extensible messaging and presence protocol (XMPP): Core*. Tech. rep. 2011.

[94] A. Salvador Palau, M. H. Dhada and A. K. Parlikad. 'Multi-agent system architectures for collaborative prognostics'. In: *Journal of Intelligent Manufacturing* 30.8 (2019), pp. 2999–3013.

[95] M. Schatten. 'BARICA-Beautiful ARtificial Intelligence Cognitive Agent'. In: (2019).

[96] M. Schatten. 'Organizational architectures for large-scale multi-agent systems' development: an initial ontology'. In: *Distributed Computing and Artificial Intelligence, 11th International Conference*. Springer. 2014, pp. 261–268.

[97] M. Schatten, T. Peharda and J. Rasonja. 'A Game Engine Layer for the Implementation of Massively Multiplayer On-line Interactive Fiction'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2021, pp. 11–16.

[98] M. Schatten, T. Peharda and I. Tomicic. 'Towards an Orchestrated Game Development Approach to Digital Twinning in Autonomous Vehicles'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2022, pp. 3–8.

[99] M. Schatten, R. Protrka et al. 'Conceptual architecture of a cognitive agent for telemedicine based on gamification'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2021, pp. 3–10.

[100] M. Schatten, I. Tomičić and B. O. Đurić. 'Orchestration platforms for hybrid artificial intelligence in computer games-a conceptual model'. In: *Central European conference on information and intelligent systems*. Faculty of Organization and Informatics Varazdin. 2020, pp. 3–8.

[101] M. Schatten et al. 'A cognitive agent's infrastructure for smart mobility'. In: *Transportation Research Procedia* 64 (2022), pp. 199–204.

[102] M. Schatten et al. 'Large-scale multi-agent modelling of massively multi-player online role-playing games–a summary'. In: *Central european conference on information and intelligent systems*. Faculty of Organization and Informatics Varazdin. 2017, pp. 193–200.

[103] M. Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.

[104] C. E. Shannon. 'XXII. Programming a computer for playing chess'. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[105] A. Singh, C. Ramakrishnan and S. A. Smolka. 'A process calculus for mobile ad hoc networks'. In: *Science of Computer Programming* 75.6 (2010), pp. 440–469.

[106] N. Singh et al. 'Load balancing and service discovery using Docker Swarm for microservice based big data applications'. In: *Journal of Cloud Computing* 12.1 (2023), pp. 1–9.

[107] K. Slonneger and B. L. Kurtz. *Formal syntax and semantics of programming languages*. Vol. 340. Addison-Wesley Reading, 1995.

[108] *Smart Python Agent Development Environment*. `https://spade-mas.readthedocs.io/en/latest/readme.html`.

[109] G. G. Smith. *iMapBook*. `https://imapbook.com/`. Web-based interactive eBook platform integrating computer games to enhance reading comprehension. 2013.

[110] R. Smith. *Docker Orchestration*. Packt Publishing Ltd, 2017.

[111] *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model overview and usage*. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Mar. 2024.

[112] I. Taleb et al. 'A Holonic Multi-Agent Architecture For Smart Grids.' In: *ICAART (1)*. 2023, pp. 126–134.

[113] J. Thönes. 'Microservices'. In: *IEEE software* 32.1 (2015), pp. 116–116.

[114] A. Trenta. 'Accounting AI Measures as ISO/IEC 25000 Standards Measures'. In: *Proceedings http://ceur-ws. org ISSN* 1613 (2023), p. 0073.

[115] V. Vaishnavi and W. Kuechler. 'Design Science Research in Information Systems'. In: *Association for Information Systems* (2004). ISSN: 02767783.

[116] J. Venable, J. Pries-Heje and R. Baskerville. 'FEDS: a framework for evaluation in design science research'. In: *European journal of information systems* 25.1 (2016), pp. 77–89.

[117] F.-Y. Wang et al. 'Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond'. In: *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016), pp. 113–120.

[118] Y.-C. Wang et al. 'An overview on generative ai at scale with edge-cloud computing'. In: *IEEE Open Journal of the Communications Society* (2023).

[119] S. Weber. 'Design science research: Paradigm or approach?' In: (2010).

[120] C. Wei, K. V. Hindriks and C. M. Jonker. 'The Role of Communication in Coordination Protocols for Cooperative Robot Teams.' In: *ICAART (2)*. 2014, pp. 28–39.

[121] D. Weyns and T. Holvoet. 'A formal model for situated multi-agent systems'. In: *Fundamenta Informaticae* 63.2-3 (2004), pp. 125–158.

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

[123] M. Wooldridge. 'Intelligent agents'. In: *Multiagent systems: A modern approach to distributed artificial intelligence* 1 (1999), pp. 27–73.

[124] Q. Wu et al. 'Autogen: Enabling next-gen llm applications via multi-agent conversation framework'. In: *arXiv preprint arXiv:2308.08155* (2023).

[125] B. Wymann et al. *TORCS: The Open Racing Car Simulator, v1.3.5*. `http://www.torcs.org`. Accessed: 2025-04-26. 2013.

[126] M. Xu et al. 'Unleashing the power of edge-cloud generative ai in mobile networks: A survey of aigc services'. In: *IEEE Communications Surveys & Tutorials* (2024).

[127] S. Zeb et al. 'Toward AI-enabled nextG networks with edge intelligence-assisted microservice orchestration'. In: *IEEE Wireless Communications* 30.3 (2023), pp. 148–156.

[128] A. Zerouali, R. Opdebeeck and C. De Roover. 'Helm charts for Kubernetes applications: Evolution, outdatedness and security risks'. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE. 2023, pp. 523–533.

[129] M. Zhang and J. Li. 'A commentary of GPT-3 in MIT Technology Review 2021'. In: *Fundamental Research* 1.6 (2021), pp. 831–833.

Appendices

Appendix A

QUESTIONNAIRES

A.1 Michael van Elk

A.1.1 Agents' orchestration specification

1. Do you find the agents' orchestration specification easy to write and understand?

   "I find the specification easy to read, understand and write, because:

   - The import syntax matches the semantics of importing in code
   - The `.` notation is an easy shorthand for a pass-through
   - There is a limited set of top-level keys (`import`, `environment`, `channel`, `agent`) and their meaning is clear at a glance
   - The use of self also has equivalent semantics to its use in programming languages

   However the meaning and use of special keywords like `ENV_INPUT` is less immediately clear. Does this reference all inputs, only those set in the environment, or something else? Of course after familiarizing with the language and syntax these things will matter less, but clarity at-a-glance is still good to aim for."

   —

2. Does reading a communication flow specification give you a clear understanding of which agents communicate and how?

   "Yes, the specification makes the communication graph explicit and therefore easy to reason about and understand. The addition of agent parametrization adds some measure of insight into how each agent performs its tasks." —

3. Do you think the programming language is type-safe enough for reliable implementation?

   "I personally did not encounter any edge cases, and the specification seems straightforward enough for effective type checking." —

4. Are the semantics of the language comprehensive enough to address real-world use cases?

   > "By not specifying the message structure, the format allows for arbitrary messages to be sent/received, which does not impose restrictions on the use cases to be implemented. It is similar in this way to an event bus. As long as communication channels can be defined beforehand, the language seems adequate to define the communication structure for a MAS. For cases requiring dynamic/runtime definition of communication channels, this limitation will make it unsuitable, but as I understand it this was an explicit design choice/goal."
   >
   > —

5. Are the supported element types (agent, channel, environment, holon, and execution flow) intuitive and descriptive enough?

   > "Agent, channel, and holon are all intuitive abstractions to me, however as described above, I do not see the need to define a separate entity (`environment`) for channels used between holons. Holons are essentially an agent 'type', as environments are essentially a channel 'type'. However, for holons a separate abstraction is useful, as the parametrization is different, whereas channels ideally would not be aware who is communicating over them." —

6. Is the parametrization of agents useful in your use cases?

   > "As a system that aims to be agnostic to agent implementation, focusing on the definition of communication topology, it makes sense to me to have some way of seeing how agents handle their messages in the orchestration specification. My only concern would be that the parametrization is not validated against the implementation (as the framework aims to be agnostic to this), meaning there may be a mismatch between specification and implementation." —

7. Which of the implemented execution flows do you find most useful?

   > "I think that with the combination of sequential, parallel and conditional all scenarios can be handled. Comparing it to Docker Compose, by default you'd want to start everything in parallel, unless an agent has a dependency on another, then you'd specify `depends_on`. Conditional execution makes less sense to me in this setting, as you'd expect all created services to be always 'alive', and if one dies and this leads to another agent starting, I imagine the communication graph would need to update, which goes against the principle that these channels are defined at the start. It makes more sense to me in a workflow-type situation, where you could use it to define control flow. However perhaps I haven't fully grokked the types of conditions and therefore scenarios this could be used, so I will caveat the above based on this." —

8. Are there execution flows missing from the current system that you think should be added?

   "No, I think sequential + parallel + conditional is enough." —

9. Is the range of supported communication protocols (TCP and UDP) sufficient to support communication between agents, environments, and channels?

   "I think the protocols cover the main ones you would expect/want in for this kind of system. The agents are able to choose their protocol, and the channels can translate to the orchestration's internal protocol. Ultimately the protocol shouldn't matter much as long as the message can be passed." —

10. Do you see the concept of holons as valuable? If so, in what ways?

    "It is useful in the sense that you can hide from other agents that an agent may interact with other agents before responding or taking further actions. Where it seems less useful to me is that channels between holons are defined as a separate entity (`environment`) rather than as a normal channel, which means the abstraction is leaky, and does not effectively hide from the rest of the system that a holon can be treated as if it was a single agent." —

11. How straightforward do you find replacing an existing agent with another within the system?

    "As the system operates at the level of interfaces and communication channels, the agent implementation is effectively hidden, so replacing one is easy." —

12. How does specifying communication flows in this language compare to other agent frameworks you have used?

    "Specifying communication graphs beforehand is an interesting approach, and makes it distinct from other frameworks I have encountered. Usually the system makes no strong assumptions about who talks to who, and agents are free to communicate with others however their implementation defines it. The ability to define and run the agents in various ways over various protocols allows for an ecosystem approach, where anyone could add any agent as long as it adheres to the interfaces. However the need to define both the communication and the specification up front requires coordination between agent developers beforehand, which makes this pattern less natural in this system." —

13. How useful do you find the channel's ability to transform messages from one format to another during transmission?

> "Having flexible mechanisms to transform messages at the communication layer can be very useful for making heterogeneous systems compatible when you don't have access to their implementation, see for example the conversion of OpenTelemetry traces in the otel-collector using the `transform:` block."
>
> —

14. Are there additional concepts you would find valuable in the language?

> "At a systems level, it is useful to have a specification of the communication channels in a MAS. It might also be useful to know what other communication takes place, for example between an agent and some external service or data source. This way the specification can move closer to defining the full 'network topology' rather than only the inter-agent communication." —

A.1.2 Orchestration platform

1. Do you see the artifact being implementable in the majority of systems?

> "I would say that the flexibility in communication protocol combined with the channel adapter pattern effectively allows the system to interoperate with other systems." —

2. What is the typical number of agents in your MAS?

> "The system should scale to any number, but common use cases require only a handful to implement." —

3. Do you think the error handling mechanism covers a good amount of edge cases (e.g., an agent goes down)?

> "The restart mechanism using the + symbol seems as much as an orchestrator can be expected to do, see Kubernetes as example which does the same. How often and when to restart are of course parameters that can be set using arbitrarily complex logic, but this is not necessary." —

4. Can intelligent agents (e.g., Gen AI-powered) be easily integrated with the platform?

> "I see no reason why they shouldn't be, as the platform does not specify anything about agent's implementation." —

5. Do you see any performance downsides in the solution?

> "I see no obvious limitations here, the communication protocols are all performant and the logic not overly complex at runtime." —

6. Is the support for distributed agents sufficient, particularly in terms of communication, synchronization, and fault tolerance?

> "I would say this is one of the main characteristics designed for in this solution, so yes." —

7. Does the cloud support (e.g., running in Docker [110]) match your needs in terms of deployment flexibility, scalability, and compatibility with your existing infrastructure?

> "Yes, the main requirement for cloud environments is containerization, and the platform supports it." —

8. Are there any additional functionalities of the orchestration platform that you would find useful?

> "Dynamic execution and communication graphs would allow the system to be useful in more scenarios. The comparison would be between Docker Compose and Kubernetes: both use static configuration files, but the presence of an API server in Kubernetes allows for dynamic discovery of other services. Perhaps a static communication / execution graph can be derived from the API server (or equivalent) at any point in time, offering its benefits while allowing a more dynamic ecosystem." —

A.1.3 Agent definition

1. Is the agent definition easy enough to write?

> "Yes, the simplicity and YAML-like syntax make it easy to write." —

2. Is the agent definition clear and accessible to non-technical stakeholders, such as project managers or domain experts?

> "It is clear to me, but being technical, I cannot speak strongly to whether it is clear to non-technical stakeholders. I expect non-technical stakeholders will be confused by the many technical terms in the agent definitions, such as `data-type, type: unix`, and values such as `<!eof!>` probably look scary to them." —

3. Are the supported properties for agent definition sufficient for your use cases? If not, what is missing?

> "I do not have a strong opinion here, as the agent definition does not limit the internal logic of the agent." —

4. How much effort is required to adjust an existing agent for inclusion into the system based on the artifact?

> "One limitation mentioned in the documentation is of concern here:
>
> *While support for different protocols on input and output provides flexibility, it adds complexity for request-response protocols. For example, HTTP expects a synchronous response to a request, but here, requests and responses may occur on different endpoints.*
>
> Agents dependent on synchronous request/response mechanisms would need to implement workarounds for this limitation, but for asynchronous / message based agents the customization does not seem too much." —

5. Do updates to your existing agent business logic after its inclusion in the artifact-based MAS require changes to the agents' orchestration specification (particularly, specification of communication flows)?

> "The same limitation for agents designed around synchronous communication patterns applies here I think. However after the initial conversion, further updates likely would not require further changes." —

6. Which communication protocols do your agents utilize for communication with external systems?

> "HTTP and SSE" —

A.2 Rio Kierkels

A.2.1 Agents' orchestration specification

1. Do you find the agents' orchestration specification easy to write and understand?

> "It is quite easy to learn, possibly because the language itself is quite small. However I don't completely see why the orchestration spec and the ad need to be in separate languages. Unless if statements and loops are introduced, it might all work from the YAML markdown language." —

2. Does reading a communication flow specification give you a clear understanding of which agents communicate and how?

> "It requires a larger amount of context to fully understand the graph. Just looking at the channels inside the body of an agent block isn't enough. Also the start sequence doesn't show how they communicate.
>
> It would help if you could pass in agents' channels as parameters while instantiating them e.g. `agent_1(agent_2.c)`. Or use pipelining like some functional languages can do like Haskell for example." —

3. Do you think the programming language is type-safe enough for reliable implementation?

   > "The types of a symbol can only be determined when reading the entire document as they might mean different things depending on their position within the specification. This makes static analysis harder along with harming readability and not providing helpful hints around the intent of the system or elements of the system." —

4. Are the semantics of the language comprehensive enough to address real-world use cases?

   > "A capability that I'm missing is the support of dynamism. Having agents/holos come and go and automatically setup and break connections will make the system a lot more suitable for a range of other applications. Not to mention the increased reliability it brings." —

5. Are the supported element types (agent, channel, environment, holon, and execution flow) intuitive and descriptive enough?

   > "The current role of environment doesn't seem to be different enough from channels to warrant their existence." —

6. Is the parametrization of agents useful in your use cases?

   > "The parametrization itself is quite useful although I'd rather have seen it be required instead of optional. For example, making passing channels as arguments required and not allowing the capture of channels from the outer scope will reduce the errors people can make." —

7. Which of the implemented execution flows do you find most useful?

   > "Decoupling the direct dependency of one agent on another by utilizing channels. This helps the resilience and composability of the system as agents don't need to explicitly support another agent. It also allows for out-of-order starting of agents, especially if the channels support some kind of buffering." —

8. Are there execution flows missing from the current system that you think should be added?

   > "A shared random access data storage pool that is accessible by all agents would be helpful. The agents could together build up a knowledge base or maybe a history of the system or subsystem they're responsible for without being locked into a streaming or request response I/O system." —

9. Is the range of supported communication protocols (TCP and UDP) sufficient to support communication between agents, environments, and channels?

> "In its current form I think it is adequate. However the system will probably benefit from adding SCTP as an additional layer 4 protocol. This is a message based protocol with characteristics of both UDP and TCP but with message boundaries built in. This removes the need for delimiter configuration as shown in the .ad files.
>
> Apart from that consider swapping out netcat for socat as a transport utility. Not only will it enable SCTP as a protocol, but many more as well. It can be the basic building block for channels with its support to hook up any program to any protocol or other program, allowing for easy transport chaining and integration of data transformation programs." —

10. Do you see the concept of holons as valuable? If so, in what ways?

> "Yes, very much so. If only to allow grouping of agents creating a compositional primitive. I am less enthusiastic about the difference in interface between holos and agents. Apart from creating collections of agents I don't see any functional difference between holos and agents. To allow for better composition the agents and holos could offer identical interfaces. Over time holos can start distinguishing themselves more by offering additional features like the aforementioned data storage component." —

11. How straightforward do you find replacing an existing agent with another within the system?

> "Very much so. This seems to be enabled by channels and their ability to do data and transport protocol transformations. This makes it a very extensible and easily adaptable system." —

12. How does specifying communication flows in this language compare to other agent frameworks you have used?

> "I have no experience with agent orchestration frameworks but from my point of view it does not really differ from normal service software orchestration. Eventually it all comes down to which interfaces do you support. Locking down your Interface Definition Language (IDL) combined with having predefined functions described in that IDL (thing request/reply messaging or key/value storage) creates a high level of interoperability between components. Orchestrating the implementations of these IDL based functionalities becomes less important and you can tackle that problem in any way you like. You can go full on kubernetes api specifications, which can be very verbose, or minimize it into its own little IDL like protobuf (which coincidentally can function as a serialization format as well)." —

13. How useful do you find the channel's ability to transform messages from one format to another during transmission?

    "Extremely useful. This is probably its biggest strength." —

14. Are there additional concepts you would find valuable in the language?

    "Explicit types in the agent parameter signatures. This will help reduce errors and will allow language expansion in the future without having to worry too much about needing to invent additional syntax. It will also allow you to remove "special" variables, like `ENV_INPUT` and `ENV_OUTPUT` since they can be passed to the agent's parameters explicitly. Types tend to make languages more expressive in their intent." —

A.2.2 Orchestration platform

1. Do you see the artifact being implementable in the majority of systems?

    "Only in systems where non-determinism and high latency are acceptable, I do see it able to be implemented. As I mentioned before, this screams classic service oriented architecture." —

2. What is the typical number of agents in your MAS?

    "Not directly applicable but I've run and designed systems with around 30-50 services." —

3. Do you think the error handling mechanism covers a good amount of edge cases (e.g., an agent goes down)?

    "I've not seen any particular mention of failure handling mechanisms apart from restarting processes. Often that's enough but because scheduling, migration and network partition tolerance have not been covered there are huge areas where it is unclear how the system behaves when faced with those types of problems." —

4. Can intelligent agents (e.g., Gen AI-powered) be easily integrated with the platform?

    "I've not seen any indication that these types of agents require special handling. Apart from generally higher latency and possibly specialized hardware requirements, these function similarly to normal networked services." —

5. Do you see any performance downsides in the solution?

    "IDL's tend to be decoupled from implementation. This allows for performance increases in different agents and different parts of the orchestrator without

> having to redesign the entire system. The one thing that might help is control around channel buffers. Network latency, slow agents or network flakiness might cause messages to pile up. Not having control over queues might mean that you'll be operating with sub par throughput." —

6. Is the support for distributed agents sufficient, particularly in terms of communication, synchronization, and fault tolerance?

> "Support for mesh-like transports would greatly benefit agent distribution capabilities. Technologies like NATS or MQTT make it easy to build reliable distributed clusters of message passing systems. The current supported transports tend to be point to point technologies excluding the underlying layer 3 IPv4 and IPv6 multicast addresses. Another helpful addition would be native support for the Cloud Events specification. This helps standardize messages of events between components." —

7. Does the cloud support (e.g., running in Docker [110]) match your needs in terms of deployment flexibility, scalability, and compatibility with your existing infrastructure?

> "Docker has little to do with cloud support. They mainly solved the distribution problem of programs which later was formalized in the OCI specification. However, because most cloud providers do have offerings that allow running container images, it is the very minimum I'd expect of a system to deliver as a distribution artifact." —

8. Are there any additional functionalities of the orchestration platform that you would find useful?

> "Interface schema definition, discoverability and validation would help a lot with preventing incompatible agents and channels to be connected to each other. Also, I've seen no mention around transport security. Finally dynamic agent creation, deletion or other topology related dynamics would help a lot in its flexibility and resilience." —

A.2.3 Agent definition

1. Is the agent definition easy enough to write?

> "Yes. Although it's not easy to see if some values are special to the definition or just strings. `<!eof!>` is a good example of this. However this might be a shortcoming of YAML not requiring strings to be quoted." —

2. Is the agent definition clear and accessible to non-technical stakeholders, such as project managers or domain experts?

> "Yes I think so. Although quite some technical knowledge is required to configure it directly. For example, non-techinical people will probably not know why or when to use the HTTP protocol vs raw TCP, what a delimiter is in the context of message parsing or what JSON is." —

3. Are the supported properties for agent definition sufficient for your use cases? If not, what is missing?

> "I'm missing message schema definitions. These help to automatically figure out if agents are compatible, both forwards or backwards." —

4. How much effort is required to adjust an existing agent for inclusion into the system based on the artifact?

> "Because most agents are integrating with hosted LLM API endpoints over HTTP. The main problem I see is the lack of request response support on a single channel. This would require quite some coordination by some kind of broker component matching up the proper responses to the proper agents originally sending out the request." —

5. Do updates to your existing agent business logic after its inclusion in the artifact-based MAS require changes to the agents' orchestration specification (particularly, specification of communication flows)?

> "Yes, see the previously mentioned point about HTTP request/response cycles." —

6. Which communication protocols do your agents utilize for communication with external systems?

> "Mainly HTTP+JSON. Sometimes newline delimited JSON." —

A.3 Igor Tomičić

A.3.1 Agents' orchestration specification

1. Do you find the agents' orchestration specification easy to write and understand?

> "Yes. The specification uses minimalistic syntax that is compact and readable (e.g., `agent a : self -> c`). However, it has a learning curve due to custom syntax and requires familiarity with concepts like channels and holons. That said, once grasped, it reduces verbosity compared to traditional agent frameworks." —

2. Does reading a communication flow specification give you a clear understanding of which agents communicate and how?

   > "Yes. The specification's structure makes flows fairly explicit: inputs, outputs, transformation paths, and startup orders are clearly defined. With small systems, this gives a complete mental model. However, for large systems with for example >30 agents, navigation might become cumbersome without visualization tools." —

3. Do you think the programming language is type-safe enough for reliable implementation?

   > "Although message formats and channels can be defined, there seems to be no static or dynamic type checking. Communication mismatches (e.g., sending XML to a JSON listener) are not flagged unless caught in transformation rules? Introducing basic typing (e.g., schemas) might improve robustness." —

4. Are the semantics of the language comprehensive enough to address real-world use cases?

   > "Mostly yes. The language covers environments, channels, agents, holons, execution flows, and transformation logic. The 4 provided demonstrations prove its capacity to handle systems with audio/video streaming, multi-agent cognition, and inter-vehicle messaging. However, dynamic runtime behaviors are currently unsupported - before you start the orchestration system, you must predefine which agents exist, how they communicate, and how they start up — all in the specification file. Once the system is running, you cannot add a new agent, change an existing agent's role, or update communication paths without restarting everything." —

5. Are the supported element types (agent, channel, environment, holon, and execution flow) intuitive and descriptive enough?

   > "Yes. The roles (agent, channel, environment, holon) are clearly separated. Each maps well to a known concept in MAS or distributed systems. Their names are intuitive and allow a beginner to quickly grasp the function of each unit." —

6. Is the parametrization of agents useful in your use cases?

   > "Yes. Agent parameterization enhances modularity and reusability. It is particularly valuable when the same agent logic is reused with different communication channels, as seen in multi-instance deployments in the MMORPG and streaming use cases." —

7. Which of the implemented execution flows do you find most useful?

   "Parallel and sequential execution flows are particularly useful for orchestrating startup orders and pipelined agent behavior. The ability to express retries (+) and error-based branching (!) also contribute to operational resilience, although limited error semantics are a bottleneck (there's no structured way to handle different kinds of failures differently, to define custom recovery strategies (e.g., restart 3 times and then alert), to route failed messages to a separate agent for inspection, to clean up or reset the system after failure (e.g., closing ports, freeing memory), or to log and report errors in a consistent way)." —

8. Are there execution flows missing from the current system that you think should be added?

   "Possibly:

   - Trigger-based execution (e.g., start agent after N messages)
   - Timeout-based execution or retries with backoff
   - Looping flows based on internal logic/state of agents

   " —

9. Is the range of supported communication protocols (TCP and UDP) sufficient to support communication between agents, environments, and channels?

   "Maybe not quite. TCP and UDP cover many use cases (especially for high- and low-latency needs), the lack of native support for pub-sub mechanisms, message queues (like MQTT, AMQP), or secure protocols (e.g., TLS over HTTP) might limit interoperability with more advanced distributed architectures." —

10. Do you see the concept of holons as valuable? If so, in what ways?

    "Yes. Holons allow for hierarchical composition of agents, which is critical in complex systems like autonomous fleets or cloud platforms. They promote encapsulation and logical separation." —

11. How straightforward do you find replacing an existing agent with another within the system?

    "Relatively easy. If the replacement agent conforms to the same communication interface (as defined in its .ad file), it can be dropped in with minimal orchestration spec change. However, lack of runtime introspection or validation means some mismatches may only manifest at runtime." —

12. How does specifying communication flows in this language compare to other agent frameworks you have used?

"APi feels good in communication clarity and lightweight syntax, compared to general-purpose MAS frameworks like JADE or SPADE. But, it seems it lacks integrated agent behaviors or reasoning models (e.g., BDI), requiring these to be handled externally or manually coded." —

13. How useful do you find the channel's ability to transform messages from one format to another during transmission?

"Yes. Transformation logic is critical when bridging heterogeneous agents." —

14. Are there additional concepts you would find valuable in the language?

"Yes. Suggested improvements:

- Dynamic agent lifecycle management (add/remove at runtime),
- Advanced triggering conditions,
- Security constructs (authentication, authorization)

" —

A.3.2 Orchestration platform

1. Do you see the artifact being implementable in the majority of systems?

"Yes - especially for research prototypes, hybrid AI systems, and cloud-native applications. The Docker-first approach and open communication standards make it flexible. Limitations might exist in integrating with legacy systems, where full control over agent startup isn't possible." —

2. What is the typical number of agents in your MAS?

"Around 150" —

3. Do you think the error handling mechanism covers a good amount of edge cases (e.g., an agent goes down)?

"Partially. Basic execution flow recovery exists (e.g., retries), but there's no full-fledged error channeling, fallback routing, or structured teardown logic. Ports may remain open." —

4. Can intelligent agents (e.g., Gen AI-powered) be easily integrated with the platform?

"Integration is protocol-agnostic and can support API-driven agents like LLMs, as seen in the B.A.R.I.C.A. case. Wrappers can encapsulate any logic, and orchestration remains unchanged as long as the communication contract is met." —

5. Do you see any performance downsides in the solution?

   "Possibly:

   - Lack of backpressure or flow control in high-throughput UDP scenarios (when messages are flying in very fast, the system seems to have no built-in way to slow things down or manage the load and some messages might be lost or cause system overload)
   - No built-in queueing or buffering?
   - No agent resource usage monitoring

   " —

6. Is the support for distributed agents sufficient, particularly in terms of communication, synchronization, and fault tolerance?

   "Relatively yes. Agents can be distributed across machines or containers, but coordination is possibly limited: there seems to be no discovery service, no centralized registry. This might limit use-cases in mission-critical or geo-distributed setups." —

7. Does the cloud support (e.g., running in Docker [110]) match your needs in terms of deployment flexibility, scalability, and compatibility with your existing infrastructure?

   "Yes, Docker and Docker Compose support enable easy local and cloud deployments." —

8. Are there any additional functionalities of the orchestration platform that you would find useful?

   "Some brainstorming here:

   - Web-based orchestration visualization and monitoring
   - Centralized logging and telemetry collection
   - Runtime validation of orchestration specs
   - Secure protocol support (TLS, OAuth)

   " —

A.3.3 Agent definition

1. Is the agent definition easy enough to write?

   "Yes, straightforward and well-documented. Developers familiar with Docker or Compose will find the structure intuitive. Templates could help reduce boilerplate further." —

2. Is the agent definition clear and accessible to non-technical stakeholders, such as project managers or domain experts?

   > "Mostly. Technical users will handle it easily. For non-technical stakeholders, a visual or form-based editor would enhance accessibility and prevent syntax errors." —

3. Are the supported properties for agent definition sufficient for your use cases? If not, what is missing?

   > "Mostly yes. Perhaps additions like retry policies, health checks, environment variables, and lifecycle hooks would improve usability." —

4. How much effort is required to adjust an existing agent for inclusion into the system based on the artifact?

   > "Low to moderate effort. As long as the agent supports one of the accepted I/O formats and can be started by the platform (UNIX, Docker, Kubernetes), inclusion mostly involves defining its .ad file. No codebase changes are typically required." —

5. Do updates to your existing agent business logic after its inclusion in the artifact-based MAS require changes to the agents' orchestration specification (particularly, specification of communication flows)?

   > "Not necessarily. If interfaces remain unchanged, the orchestration spec probably doesn't need updating. However, changing communication format or channel names might require updates. Decoupling is good, but could be improved with interface abstraction layers." —

6. Which communication protocols do your agents utilize for communication with external systems?

   > "NETCAT, HTTP." —

Curriculum Vitae

Tomislav Peharda was born on July 9, 1995, in Varaždin. He participated in high school from 2010 to 2014 at Electromechanical School for Computer Technician. In 2014 he enrolled to bachelor degree at Faculty of Organization and Informatics, Varaždin for major Information Systems. Upon successful completion, he continued master at the same Faculty, in major Information and software Engineering. In 2018, he received dean's award for best GPA in his class. In 2019, he graduated with thesis with title Developing an Intrusion Detection System using Deep Learning. In 2020, he started his academic career at Faculty, as part of a project IRI HYPER, and then throughout worked on other projects such as O-HAI 4 Games, Full STEAM ahead!, and Vidi-π. In 2020, Tomislav enrolled in a PhD program and had his thesis application approved in 2024, under the title 'Programming Language for Communication Flows Specification in Multi-Agent Systems,' with Full Professor Markus Schatten as his mentor. Tomislav has always had big interest in computer science, and specifically fields of Artificial Intelligence, machine learning, IoT, game development, as well as security. Aside from working on Faculty projects, Tomislav is also an assistant who teaches Declarative programming, Multi-agent systems, and Theory of Databases, as part of laboratory sessions. So far, Tomislav has published over 10 scientific papers.

Published research

[1] B. O. Đuric, T. Peharda and P. Martí. 'Towards a gamified system to influence behaviour of users in the context of smart mobility'. In: *34th International Scientific Conference Central European Conference on Information and Intelligent Systems: Proceedings 2023*. Varaždin: Faculty of Organization and Informatics, University of Zagreb ... 2023.

[2] B. O. Đuric et al. 'Towards Identifying Social Gamification Techniques Applicable to Groups, Applied to On-Demand Shared Transportation'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2024, pp. 1–8.

[3] S. Markus et al. 'An Orchestrated Game Streaming System for Transport and Mobility Research'. In: *Transportation research procedia* 73 (2023), pp. 126–132.

[4] T. Peharda, B. Okreša Đurić and I. Tomičić. 'Towards Application of Programming Language for Communication Flows Specification in Multi-agent Systems on Real-World Use Cases'. In: *34th International Scientific Conference Central European Conference on Information and Intelligent Systems: Proceedings 2023*. Varaždin: Faculty of Organization and Informatics, University of Zagreb ... 2023, pp. 17–22.

[5] M. Schatten, B. O. Đurić and T. Peharda. 'An agent-based game engine layer for interactive fiction'. In: *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer. 2021, pp. 385–389.

[6] M. Schatten, B. O. Đurić and T. Peharda. 'A cognitive agent for university student support'. In: *2021 IEEE Technology & Engineering Management Conference-Europe (TEMSCON-EUR)*. IEEE. 2021, pp. 1–6.

[7] M. Schatten, T. Peharda et al. 'Towards a Streamed Holographic 3D Game Engine'. In: *Central European Conference on Information and Intelligent Systems*. Varaždin: Faculty of Organization and Informatics, University of Zagreb ... 2022, pp. 17–22.

[8] M. Schatten, T. Peharda and B. O. urić. 'Towards a Communication Specification Language for Heterogeneous Service Orchestration Based on Process Calculus and

Holonic Multi-agent Systems'. In: *International Conference on Intelligent Data Engineering and Automated Learning*. Springer. 2024, pp. 202–213.

[9] M. Schatten, T. Peharda and J. Rasonja. 'A Game Engine Layer for the Implementation of Massively Multiplayer On-line Interactive Fiction'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2021, pp. 11–16.

[10] M. Schatten, T. Peharda and I. Tomicic. 'Towards an orchestrated game development approach to digital twinning in autonomous vehicles'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2022, pp. 3–8.

[11] M. Schatten et al. 'A cognitive agent's infrastructure for smart mobility'. In: *Transportation Research Procedia* 64 (2022), pp. 199–204.

[12] I. Tomičić, T. Peharda and A. Bernik. 'An Active Game Bot Detection with Security Bots'. In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2021, pp. 25–31.

[13] M. Tomičić Furjan, B. Okreša Djuric and T. Peharda. 'Improving Student Mobility Through Automated Mapping of Similar Courses'. In: *Domain-Specific Conceptual Modeling: Concepts, Methods and ADOxx Tools*. Springer, 2022, pp. 503–520.