

Bachelor Thesis
Documentation

A Multi-Agent Harness for Enterprise Workflow Automation

Semester: Spring 2026



OST

Eastern Switzerland
University of Applied Sciences

Date: 2026-06-11

Project Team: Arnel Veladzic
Fabio Gomes Silva

Project Advisor: Prof. Dr. Mitra Purandare
Co-Examiner: Dr. Marc Rösli

Part I

Abstract

Abstract

Atfinity’s Case Management System is a bank-configurable platform in which each institution defines its own schema of ontologies, processes and fields, with field visibility depending on previously submitted values; this work is grounded in a real-world configuration based on the Bank of Luxembourg’s deployment, comprising approximately 25 000 fields across 23 ontologies and 30 processes. A preceding study thesis showed that LLM-driven extraction via the Model Context Protocol (MCP) is feasible for simple, single-party onboarding, but left three issues open: the single-agent design misattributed fields across multi-party cases, only one language model was evaluated and the four hand-crafted test scenarios provided insufficient statistical grounding.

The present work addresses these issues with the Multi-Agent Preemptive Single-Shot architecture. It assigns one dedicated Ontology Agent to each party type and each agent works only on its own entity’s data and schema. Because the agents do not share context, they can run in parallel without confusing one entity’s fields with another’s. The Orchestrator assembles their outputs into a complete `predefined_objects` payload and submits it to Atfinity in a single atomic request, eliminating the need to map runtime-assigned instance identifiers to individuals, the root cause of multi-party failures in prior architectures. A provider-agnostic LLM layer (LiteLLM) lets the model behind each agent group, such as the Ontology Agents, be exchanged through a dedicated environment variable, enabling systematic model comparison without code changes. An autonomous self-healing retry loop resolves Atfinity validation errors without user intervention, correcting structural errors inline and routing semantic errors back to the responsible agent for re-extraction, with up to five attempts before failure is surfaced. Finally, a preview-and-approval workflow shows the extracted data, the matched process and the planned entity structure to the Relationship Manager before any case is committed to Atfinity.

Evaluation used a deterministic, ground-truth-driven framework with twelve test cases derived from three validated Atfinity cases across four conversational archetypes (structured, messy, self-correcting dictation and email), assessing four language models in uniform-model configurations. GPT-5.4 leads the field: an overall average F1-score of 96.25%, all NFR thresholds met (precision $\geq 90\%$, recall $\geq 90\%$, F1 $\geq 85\%$, latency ≤ 120 s), 99.25% F1 on the six-role trust structure, almost no self-healing retries (no correction cycle in ten of the twelve test cases), end-to-end latency of 25–41 s and an operational cost of \$0.94 to \$3.60 per case that scales with the number of parallel Ontology Agent invocations. Claude Sonnet 4.6 costs as much or more per case yet trails far behind, with an overall F1 of only 65.1% driven by a severe precision-recall imbalance, precision near 100% but recall collapsing to 21% on the trust case, showing that a higher price tag does not translate into better extraction quality. Claude Haiku 4.5 suffers three complete pipeline failures under high field density and peaks at 75.5% F1 and GPT-5.4-mini fails to produce any valid Atfinity submission at all due to a systematic field-value type mismatch that the self-healing loop cannot resolve.

The results confirm that the multi-agent, preemptive single-shot architecture is viable for production-grade, multi-party bank client onboarding via MCP, provided a sufficiently capable language model is assigned to the Ontology Agents. Two directions remain for future work: evaluating mixed-model configurations that assign smaller models to lower-complexity roles to reduce operational cost without sacrificing accuracy and adding an Explainability Agent that attaches per-field extraction reasoning to the preview output to further increase Relationship Manager confidence.

Part II

Management Summary

Management Summary

This management summary provides a business-level overview of the bachelor thesis for a non-technical audience. The Background and Motivation section explains the problem addressed. The Approach and Technologies section describes what was built and how. The Results section presents the outcomes and a brief outlook.

Atfinity

Atfinity is a Swiss fintech company based in Zurich that offers an AI-supported, no-code automation platform for banks and financial institutions. The platform digitalises and streamlines key compliance processes, including client onboarding, Know Your Customer (KYC), Anti-Money Laundering (AML) and client lifecycle management, while integrating with existing banking IT systems. Atfinity is used by clients including Bitcoin Suisse, Kaleido and Kaiser Partner.

Background and Motivation

Client onboarding in banking is a complex, regulated process. Relationship Managers must collect and record large volumes of client data field by field, a task that is time-consuming, error-prone and costly. A preceding study thesis showed that an AI assistant can read a client description written in plain language and automatically fill in the correct fields in Atfinity's system. That proof of concept worked for simple cases involving a single client. However, real-world onboarding often involves multiple parties in different roles, for example an account holder, a beneficial owner and a signatory all within the same case and that complexity was not addressed.

The present bachelor thesis closes this gap. The goal was to build a system capable of handling the full complexity of real banking onboarding: multiple parties, multiple roles and the complete field catalogue of a real bank deployment, in this case based on the configuration of the Bank of Luxembourg.

Approach and Technologies

The central challenge of multi-party onboarding is ensuring that each piece of client information is assigned to the correct person. A single AI model processing all parties at once risks mixing up data between individuals, a compliance risk that cannot be tuned away. The solution developed in this thesis assigns a dedicated AI agent to each type of party in the case. Every agent focuses exclusively on the data for its assigned entity and runs in parallel with the other agents. Figure 1 illustrates the end-to-end interaction between the Relationship Manager, the Multi-Agent System and Atfinity.

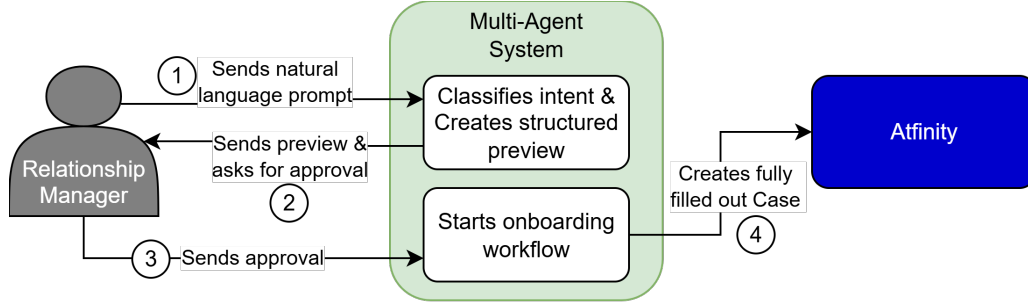


Figure 1: How it Works.

The Relationship Manager sends a plain-language prompt (1). The system classifies the intent, runs the extraction pipeline and returns a structured preview for approval (2). After the Relationship Manager confirms (3), the system creates the fully populated case in Atfinity in a single atomic submission (4). Beyond the core workflow, two additional capabilities were built into the system:

- **Automatic error recovery:** If Atfinity rejects a submission, the system identifies which field caused the problem, attempts a correction and resubmits automatically, up to five times, without requiring any manual intervention.
- **Model flexibility:** The system can work with any major AI provider. Switching models requires no code changes, which made it possible to compare multiple AI models systematically within the same evaluation setup.

Results

The system was successfully deployed and evaluated against three onboarding scenarios drawn from real Atfinity cases: a joint account with three co-holders, a data-intensive individual account and a complex trust structure with six parties in distinct roles.

System Capabilities

A Relationship Manager can now manage the entire onboarding lifecycle through plain-language conversation, creating cases, updating fields, adding or removing parties, querying open fields and discovering available processes, without needing to know any of Atfinity’s internal identifiers or field keys. All defined user stories and functional requirements were fully delivered.

AI Model Comparison

Four AI models were compared: GPT-5.4 and GPT-5.4-mini from OpenAI and Claude Sonnet 4.6 and Claude Haiku 4.5 from Anthropic. For each model, Figure 2 shows the overall average precision, recall and F1-score across all tested ground-truth scenarios. Precision is the share of extracted fields that were correct, recall is the share of the required fields that were actually found and the F1-score combines both into a single accuracy measure.

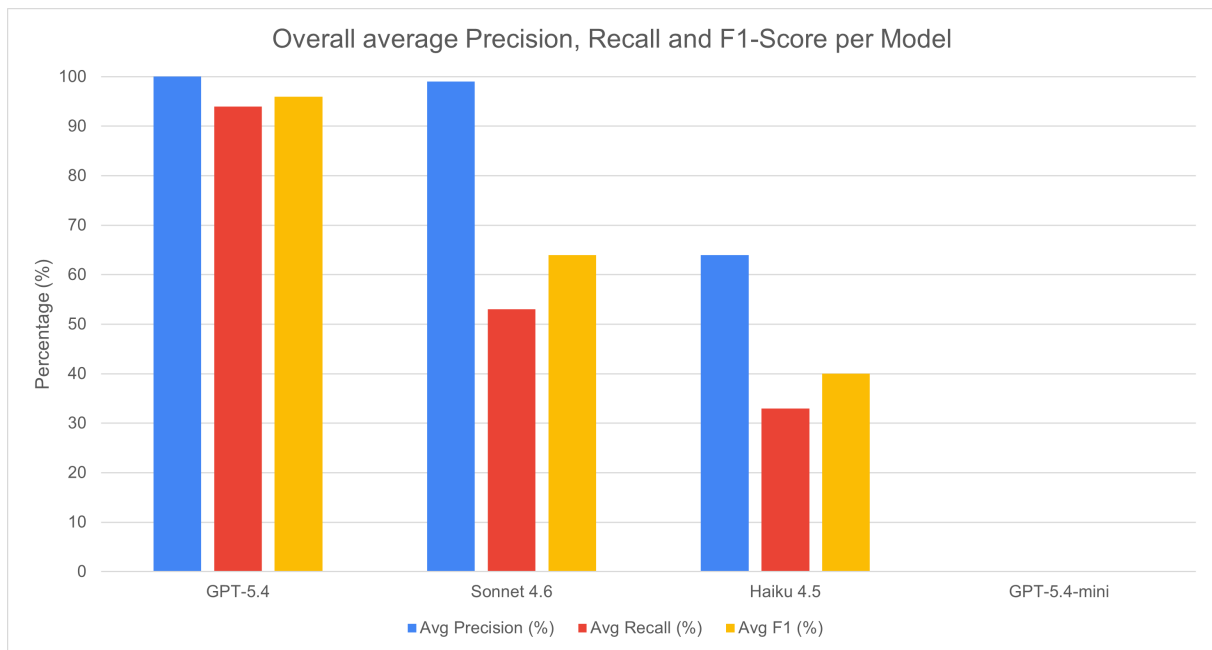


Figure 2: Overall average precision, recall and F1-score per AI model, aggregated across all tested ground-truth scenarios.

GPT-5.4 is the clear recommendation for production use. As Figure 2 shows, it combines near-perfect precision (around 100%) with high recall (around 94%) and it is the only model that meets all defined quality and performance thresholds across every scenario tested:

- **99.6%** average precision and **94.3%** average recall across all twelve test cases
- **99.25%** F1-score on the most complex trust case with six simultaneous parties
- Zero failures: every test case produced a valid, accepted Atfinity submission
- Full onboarding completed in **25 to 41 seconds**
- Despite being the most capable model, GPT-5.4 costs **less per case than Claude Sonnet 4.6**

Claude Sonnet 4.6 reaches a similarly high precision (around 99%), so the field values it does extract are almost always correct, but its recall collapses to about half (around 53%), confirming that it misses roughly half of the required fields and entities, especially in multi-party and trust scenarios. Claude Haiku 4.5 scores low on both dimensions (around 64% precision and 33% recall) and, as the cheapest option, additionally failed to produce any output at all in the most data-intensive scenario. GPT-5.4-mini registers no measurable values in Figure 2 at all, because it could not generate a single valid submission throughout the entire evaluation.

Outlook

The results confirm that AI-driven multi-party onboarding is viable and ready for the next stage of development. The most impactful next steps are:

- **Cost optimisation:** Using a less expensive AI model for the simpler coordination tasks while reserving the full GPT-5.4 model for data extraction, a switch the current architecture already supports without code changes.
- **Transparency:** Extending the preview to include a plain-language explanation of why each field value was extracted, giving Relationship Managers greater confidence in the system's decisions.
- **Production hardening:** Adapting the system for a live banking environment, including security, monitoring and integration with production infrastructure.

Contents

I	Abstract	i
II	Management Summary	iii
III	Introduction	1
1	Overview	2
1.1	The Atfinity Platform and Its Data Model	2
1.1.1	CMS Data Model and Terminology	2
1.2	Background and Problem Statement	3
1.2.1	The Client Onboarding Challenge	3
1.2.2	The AI Vision and Its Technical Obstacles	4
1.2.3	Prior Work: The Study Thesis	4
1.3	State of the Art	5
1.3.1	LLM-Based Extraction and Tool Integration	5
1.3.2	Model Context Protocol: Open Questions in Enterprise Adoption	5
1.3.3	Multi-Agent Systems and Model Trade-offs	6
1.4	Ethical and Regulatory Considerations	6
1.4.1	High-Risk Classification and Human Oversight	6
1.4.2	Data Protection, Residency and Outsourcing Compliance	6
1.4.3	Accountability and the Audit Trail	6
1.5	Goals	7
1.5.1	Product Objectives	7
1.5.2	Research Questions	7
1.5.3	Research Value	8
1.6	Deliverables	8
1.7	Abbreviations	8
IV	Product Documentation	10
2	Requirements	11
2.1	Use Case Overview	11
2.2	User Stories	12
2.2.1	User Story 1: Create Automatic Client Onboarding	12
2.2.2	User Story 2: Preview and Approve Case Before Submission	12
2.2.3	User Story 3: Ask for Still Open Fields	13
2.2.4	User Story 4: Update Existing Case	13
2.2.5	User Story 5: Get Relevant Information and Ask Questions About a Case	13
2.2.6	User Story 6: Add or Remove a Party from an Existing Case	13
2.2.7	User Story 7: Manage the Roles of a Party	13
2.2.8	User Story 8: Discover Available Processes and Workflows	13
2.3	Functional Requirements	13
2.3.1	FR1: Retrieve Case Information and Answer Questions	13
2.3.2	FR2: Extract Structured Data from Free-Text Input	13
2.3.3	FR3: Adapt to Atfinity's Incremental Field Architecture	13

2.3.4	FR4: Detect and Route User Intent	14
2.3.5	FR5: Expose All Operations as MCP Tools	14
2.3.6	FR6: Preview and Approval Workflow	14
2.3.7	FR7: Party Instance Management	14
2.3.8	FR8: Role Management	14
2.3.9	FR9: Process Discovery	15
2.3.10	FR10: Autonomous Self-Healing on Submission Errors	15
2.4	Non-Functional Requirements	15
2.4.1	NFR-1: Performance	15
2.4.2	NFR-2: Precision	15
2.4.3	NFR-3: Recall	15
2.4.4	NFR-4: F1-Score	16
2.4.5	NFR-5: Efficiency	16
2.4.6	NFR-6: Usability	16
3	Model Context Protocol (MCP)	17
3.1	Definition and Purpose	17
3.2	Architecture and Communication	18
3.3	Exposed Capabilities	18
3.4	Relevance to This Project	19
4	Multi-Agent Systems	20
4.1	Definition and Motivation	20
4.2	Agent Roles and Specialisation	20
4.3	Control Topologies	21
4.3.1	Centralised	21
4.3.2	Decentralised	21
4.3.3	Hierarchical	21
4.4	Execution Patterns	21
4.4.1	Sequential	21
4.4.2	Parallel	22
4.4.3	Hybrid	22
4.5	LLM-Based Agents	22
4.6	Relevance to This Project	22
5	Design / Architecture	23
5.1	Preamble	23
5.2	System Context Diagram	23
5.3	Architectural Overview	25
5.3.1	Study Thesis Baseline	25
5.3.2	Increased Complexity in the Bachelor Thesis	25
5.4	The Iterative Approach: Not Feasible in the Bachelor Thesis Context	25
5.5	The Hybrid Approach: Not Feasible in the Bachelor Thesis Context	26
5.6	Multi-agent Preemptive Single-Shot Architecture	27
5.6.1	Architectural Overview	27
5.6.2	Deployment Environment	27
5.6.3	Container Diagram	28
5.6.4	Components	30
5.6.5	Context Scoping per Agent	35
5.6.6	Case Creation Workflow	36
5.6.7	Quality Attributes	48
5.7	Architectural Decision Records	50
5.7.1	ADR01: OST VM as Exclusive Deployment Target with Automated CI/CD Pipeline	50
5.7.2	ADR02: Predefined-Objects Single-Shot Payload Strategy	51
5.7.3	ADR03: LiteLLM as Provider-Agnostic LLM Interface	53
5.7.4	ADR04: Orchestrator as Central Coordination Component	54
5.7.5	ADR05: Multi-Agent Decomposition with One Ontology Agent Per Ontology	55
5.7.6	ADR06: OntoDB as Shared Schema Persistence Layer	56
5.7.7	ADR07: Dynamic Ontology Agent Deployment via Docker Socket	58

5.7.8	ADR08: Hybrid Sequential-Parallel Pipeline Execution	59
5.7.9	ADR09: Self-Healing Retry Loop for Atfinity Validation Errors	60
5.7.10	ADR10: Schema Learning via SchemaDB	61
5.7.11	ADR11: Two-Stage Prompt Scoping for Multi-Party Extraction	62
5.7.12	ADR12: Preview Approval Gate with Pipeline Result Caching	63
5.7.13	ADR13: LibreChat as Frontend and MCP Client	65
5.7.14	ADR14: Caddy as Reverse Proxy	66
6	Implementation	67
6.1	Technology Stack and Dependencies	67
6.2	Schema Synchronisation	68
6.3	In-Memory Context Stores	68
6.4	LLM Integration via LiteLLM	69
6.5	Schema-Aware Field Extraction	69
6.5.1	Schema Chunking	70
6.5.2	Partial JSON Recovery	70
6.6	Preview Cache	71
6.7	Self-Healing Submission Loop	71
6.8	Schema Learning	72
6.9	Dynamic Agent Deployment	72
6.10	CI/CD Pipeline	73
6.10.1	SSH Helper and Secret Injection	73
6.10.2	Deploy Stage Mechanics	74
6.10.3	Evaluate Stage	74
6.11	Testing Framework	74
6.11.1	Experimental Architecture and Environment	75
6.11.2	Mathematical Formulation of Metrics	76
6.11.3	Token Usage, Cost and Retry Tracking	78
6.11.4	Stochastic Repetition and Variance Measurement	79
6.11.5	Individual Agent Evaluation	79
7	Results	80
7.1	Achievement of Project Goals	80
7.2	User Story Coverage	81
7.3	Functional Requirements	82
7.4	Non-Functional Requirements	83
7.5	Implementation Status of Requirements	84
7.6	Illustrative Onboarding Scenario	85
7.7	Research Question Results	88
7.7.1	RQ1, Accuracy and Robustness: How does multi-agent decomposition affect ex- traction correctness and workflow reliability?	88
7.7.2	RQ2, Efficiency: How do agent architecture and LLM size affect runtime, LLM call count and operational cost?	89
7.7.3	RQ3, Agent Specialisation: Does assigning different LLMs to specialised agents improve performance or reduce cost compared to a uniform-model configuration? .	90
7.7.4	RQ4, Explainability: How can an Explainability Agent improve interpretability and trust in the system's outputs?	90
7.7.5	RQ5, Architecture Necessity: What structural constraints of multi-party bank client onboarding require a multi-agent design and is the operational cost justi- fied against the manual alternative?	91
7.8	Evaluated Model Configurations	93
7.9	LLM Evaluation	93
7.9.1	GPT-5.4: Benchmark Performance	94
7.9.2	Claude Sonnet 4.6: Performance Analysis	97
7.9.3	Claude Haiku 4.5: Performance and Critical Failures	100
7.9.4	GPT-5.4-mini: Complete Pipeline Rejection	103
7.10	Cross-Model Comparison	105
7.10.1	Precision and Recall Profile	107
7.10.2	Reliability and Schema Compliance	110

7.10.3	Cost and Operational Analysis	112
7.11	Limitations	118
7.12	Delivered Artefacts	119
V	Project Documentation	120
8	Project- and Time Management	121
8.1	Project Planning	121
8.1.1	Methodology	121
8.1.2	Project Members	121
8.1.3	Responsibilities	122
8.1.4	Collaboration and Meetings	122
8.1.5	Planning Tools	122
8.1.6	Long-Term Planning	123
8.1.7	Milestones	125
8.1.8	Short-Term Planning	126
8.2	Time Management	127
8.2.1	Time Management Statistics	127
8.3	Risk Management	129
8.3.1	Risks and Mitigations	129
8.3.2	Risk Matrix in Sprint 1	131
8.3.3	Risk Matrix in Sprint 2	133
8.3.4	Risk Matrix in Sprint 3	135
8.3.5	Risk Matrix in Sprint 4	137
8.3.6	Risk Matrix in Sprint 5	139
8.3.7	Risk Matrix in Sprint 6	141
8.3.8	Risk Matrix in Sprint 7	143
8.3.9	Risk Matrix in Sprint 8	145
9	Personal Reports	147
9.1	Arnel Veladzic	147
9.1.1	What Went Well	147
9.1.2	Areas for Improvement	147
9.1.3	Personal Highlights	147
9.2	Fabio Gomes Silva	148
9.2.1	What Went Well	148
9.2.2	Areas for Improvement	148
9.2.3	Personal Highlights	148
	Bibliography	148
	Appendices	155
A	Examples of Working Client Onboardings	156
A.1	Example of a Full Client Onboarding	156
A.2	Mermaid Source Code - Full Sequence Diagram	164

Part III

Introduction

Chapter 1

Overview

This chapter introduces the technical and organisational context of the bachelor thesis. Section 1.1 introduces the Atfinity platform, its bank-configurable data model and the core CMS terminology used throughout this thesis. Section 1.2 describes the client onboarding challenge in the banking sector, the AI vision motivating this project and the structural limitations of the preceding study thesis that define the starting point for the present work. Section 1.3 surveys the relevant state of the art across LLM-based extraction, tool integration, the Model Context Protocol and multi-agent systems. Section 1.4 documents the ethical and regulatory constraints applicable to an AI system deployed in a banking compliance context. Section 1.5 defines the product objectives. Section 1.6 lists the project deliverables.

1.1 The Atfinity Platform and Its Data Model

Atfinity (Atfinity AG) is a Swiss financial technology company based in Zurich. It provides a no-code process automation platform for banks and financial institutions, enhanced with AI capabilities. The platform allows organisations to digitise and automate complex operational workflows without writing custom code. Supported processes include client onboarding, Know-Your-Customer (KYC), Know-Your-Business (KYB), Anti-Money Laundering (AML) checks, client lifecycle management and loan origination. The platform integrates with existing bank infrastructure and is used by institutions such as Bitcoin Suisse, Kaleido and Kaiser Partner. [4]

A defining feature of Atfinity’s architecture is its bank-configurable data model. Each institution defines its own data fields, validation rules, permitted values and conditional dependencies within a JSON-based schema. This allows Atfinity to serve banks with different regulatory requirements and internal processes without modifying the platform’s code for each deployment. As a consequence, Atfinity does not expose a fixed, uniform data model. The schema varies per institution and this variability has direct implications for any AI system that must interact with it.

1.1.1 CMS Data Model and Terminology

The following concepts are fundamental to Atfinity’s CMS and recur throughout this thesis. They are defined here in the order they build on each other so that subsequent chapters can use them without repeated explanation.

Case

A case is the top-level record in Atfinity that represents a single onboarding engagement. It groups all involved parties, their associated field data and the process under which they are being onboarded.

Ontology

An ontology is a named entity type within Atfinity’s data model. Each ontology defines a structured set of fields that describe one category of party or object in a banking context. Typical ontologies include Person, Company, Account and Fund. The exact set of ontologies and the fields within each are defined by the bank, meaning they vary per institution.

Field

A field is a single data attribute within an ontology. Each field carries a key, a data type (such as string, integer, date or enumeration), optional validation constraints and, for enumeration fields, a fixed set of permitted values. Fields may also carry conditional dependencies: a field only becomes visible in the Atfinity UI after a specific value has been entered in another field. Importantly, these dependency rules are not exposed through Atfinity’s schema API. It is therefore not possible to determine programmatically which field value triggers the visibility of which dependent field; the dependency graph can only be observed at runtime by inspecting which fields Atfinity reveals after each submission.

Instance

An instance is a concrete occurrence of an ontology within a case. Where an ontology defines the schema, an instance holds the actual field values for one specific party. A single case may contain multiple instances of the same ontology, for example two Person instances representing two different natural persons involved in the same onboarding.

Role

A role defines the function that an instance plays within a case. Typical roles include Account Holder, Beneficial Owner, Authorised Signatory and Controlling Person. A single instance may carry more than one role if the same party fulfils multiple functions in the case.

Process

A process is a workflow template that defines the structure of a case. It specifies which ontology types must be present and which roles each instance must fill. The set of available processes is bank-specific, just as the ontology schemas are.

Outcome Ontology

Every process designates one ontology as its outcome ontology. This is the primary entity that Atfinity creates automatically when a case is opened, serving as the structural backbone to which all other instances are linked. For a standard private banking onboarding, the outcome ontology is Account.

1.2 Background and Problem Statement

The following subsections establish the problem context from three perspectives. The client onboarding challenge describes the regulatory requirements and operational complexity that make bank client onboarding a time-consuming manual process. The AI vision outlines the automation approach Atfinity envisions and the three technical obstacles this system must overcome. The prior work section summarises the preceding study thesis, its key findings and the structural limitations that define the starting point for the present work.

1.2.1 The Client Onboarding Challenge

Client onboarding in the banking sector is governed by strict regulatory requirements. These requirements determine which data a bank must collect and verify for each client, depending on the client’s profile and the type of account being opened. In Switzerland, the relevant legal framework includes the Anti-Money Laundering Act (AMLA)¹ enforced by the Swiss Financial Market Supervisory Authority (FINMA), which requires Know-Your-Customer (KYC) and AML due diligence before a client can be accepted. At the international level, the Financial Action Task Force (FATF) sets the risk-based standards on which these national rules are built. [8]

Not all data fields apply to every client. Which fields are required depends on the client’s individual circumstances. For example, if a client is identified as a Politically Exposed Person (PEP) or is associated with a high-risk jurisdiction, enhanced due diligence is triggered, requiring additional documentation

¹Bundesgesetz über die Bekämpfung der Geldwäscherei und der Terrorismusfinanzierung (GwG), SR 955.0.

and compliance checks. The situation becomes more complex in multi-party cases, where an account involves several natural persons and legal entities in distinct roles, such as Account Holder, Beneficial Owner or Authorised Signatory. Each party in such a case must be evaluated independently, meaning the set of applicable fields differs per party.

Atfinity handles this complexity by encoding the conditional business rules directly into its data model. When a Relationship Manager (RM) opens a new client case, only a small initial set of fields is shown. As the RM fills in those fields, the platform evaluates the entered values and reveals additional dependent fields accordingly. For example, the field Total Wealth only becomes visible after the Account Type has been selected. This dynamic approach reduces the cognitive load on the RM and ensures schema consistency. However, the data entry process itself remains manual and time-consuming.

1.2.2 The AI Vision and Its Technical Obstacles

To reduce this manual workload, Atfinity envisions an AI-driven extension to its platform. The goal is to allow Relationship Managers to describe client information in plain, unstructured text, as they would in a conversational message. An AI system would then read that text, extract the relevant data and automatically create and populate a new onboarding case in Atfinity with the correct field values. This would replace a largely manual, error-prone process with an automated one.

Building such a system presents three distinct technical obstacles. The first arises from Atfinity’s configurable architecture. Because each bank defines its own schema, a solution that hard-codes field mappings for one bank would not work for another and any schema change would require manual re-engineering of the extraction logic. The system must therefore read and interpret the active schema at runtime, without prior knowledge of its structure.

The second obstacle is the conditional structure of Atfinity’s field schema. Not all fields are visible at the outset of a case; instead, fields are progressively revealed as earlier fields are filled in, based on the bank’s configured dependency rules. Systems that interact with Atfinity incrementally must therefore manage multiple submission cycles, with each cycle potentially exposing new fields that the previous submission activated. A preemptive approach addresses this differently: the complete field definitions for each ontology can be retrieved from Atfinity’s REST endpoints independently of any active case, as it is not tied to case state. By retrieving those definitions in advance and assembling a complete `predefined_objects` payload before any case is created, the multi-cycle workflow is bypassed entirely. The conditional field activation then occurs implicitly on Atfinity’s side as it processes the submitted payload.

The third obstacle concerns how Atfinity identifies each party within a case and why this makes multi-party onboarding structurally difficult for any incremental approach. When a case is created, Atfinity assigns a unique numeric identifier to each party instance involved in that case. Every field value belonging to a party must then be submitted under a composite key of the form `<instanceId>-<fieldId>`, where the instance identifier is the one Atfinity generated. Critically, these identifiers are assigned by the server at the moment of case creation and cannot be known beforehand.

The problem becomes more acute because Atfinity does not wait passively for the user to define how many parties are involved. When a specific business partner type is selected during case creation, Atfinity automatically generates a set of party instances based on the compliance rules of that onboarding type. A trust onboarding, for example, may cause Atfinity to immediately create instances for a trustee, a settlor and a beneficial owner, each pre-assigned its required role. The number and structure of these auto-generated instances is determined by the bank’s configured compliance rules, not by the natural language description provided by the Relationship Manager. The system therefore cannot predict in advance how many instances will be created, nor can it determine which auto-generated instance should receive the data of which individual from the input prompt.

1.2.3 Prior Work: The Study Thesis

The preceding study thesis, *Conversational AI for Bank Client Onboarding*, demonstrated that this vision is technically feasible. Three single-agent prototype architectures were implemented and evaluated on a shared set of onboarding scenarios. Each variant used a different strategy for managing how often the

LLM was invoked during case processing. The iterative approach triggered a dedicated LLM call at each update cycle to extract the newly visible fields. The hybrid approach combined a single upfront field mapping with an iterative LLM fallback for fields not covered by the initial mapping. The preemptive approach generated the complete field mapping in one LLM call before case creation, with no further model invocations during processing. All three variants ran on a shared containerised infrastructure that exposed Atfinity’s CMS as an MCP server and used Claude as the reasoning model. The evaluation confirmed that LLM-driven extraction and automated case submission via MCP is viable for simple onboarding scenarios and identified a trade-off between processing speed and extraction completeness across the three approaches.

The study thesis also revealed clear structural limitations. First, all three architectures failed on multi-party onboarding scenarios. Because a single agent handled all field extraction within one shared reasoning context, it frequently assigned field values to the wrong party and omitted fields that should have been filled. Second, the single-agent design had no mechanism for separating the reasoning for one entity type from another. As the number of parties in a case increased, so did the complexity of the task, with no structural boundary to keep the entities apart. Third, the prototypes could not manage the creation, sequencing and linking of multiple party instances within a single case, a requirement of Atfinity’s data model that the flat field-mapping approach could not satisfy. Fourth, the evaluation relied on a small number of manually constructed test cases. This limited the repeatability of the results and prevented systematic coverage of multi-party scenarios, corner cases and ambiguous or incomplete inputs that arise in real-world onboarding. These limitations define the starting point for the present work and the research directions addressed in this thesis, formulated as research questions in Section 1.5.

1.3 State of the Art

The following subsections survey the three technical areas most directly relevant to this work: LLM-based extraction and tool integration, the Model Context Protocol and multi-agent systems with model trade-offs.

1.3.1 LLM-Based Extraction and Tool Integration

Instruction-tuned large language models can extract structured data directly into a JSON schema without task-specific training, using zero-shot or few-shot prompting alone [16]. Standard extraction benchmarks use fixed, pre-defined schemas with a known set of entity types. They provide no evaluation signal for scenarios where the schema is unknown at inference time and must be read from a live configuration. In Atfinity’s case, each bank defines its own schema, so the model cannot rely on prior knowledge of the field structure. It must infer the meaning of each field and the appropriate value to assign from the schema description alone at runtime. This requirement shapes the core extraction design of the system developed in this thesis.

Connecting LLMs to external systems requires tool use, also referred to as function calling. This mechanism works well for straightforward integrations but becomes unreliable when a workflow involves multiple dependent steps, conditional logic and transactional requirements. Standard tool use provides no means of preserving intermediate state between sequential calls, no transaction semantics to recover from partial failures and no built-in mechanism to act on validation errors returned by the external system between calls. In the Atfinity onboarding workflow, a single case creation involves multiple ordered API calls where each step depends on the outcome of the previous one and on runtime validation feedback from the schema. These gaps are directly relevant to the design of this work.

1.3.2 Model Context Protocol: Open Questions in Enterprise Adoption

The Model Context Protocol (MCP), published by Anthropic in November 2024, is an open standard for connecting LLMs to external tools and data sources [12]; a detailed technical treatment is provided in Chapter 3. Guidance on using MCP in enterprise banking environments is limited. There are no established practices for handling failed tool calls with retry logic, managing transaction boundaries across sequences of dependent tool calls or partitioning context between agents with different roles in a multi-agent setup. The present work addresses these gaps in a concrete industrial setting.

1.3.3 Multi-Agent Systems and Model Trade-offs

Multi-agent systems address complex tasks by distributing work across multiple specialised agents. For structured data extraction across multiple entity types, the decomposition argument is direct: if a single agent handles all parties at once, entity boundaries remain implicit within one shared context, increasing the risk of cross-entity field attribution errors. Assigning a dedicated agent to each entity type creates an explicit structural boundary. Published evaluations of multi-agent approaches focus primarily on open-ended reasoning and code-generation tasks [19]. Structured extraction in schema-driven workflows, where the sole success criterion is correctness against a fixed schema, is not well represented in the literature. The present work provides empirical evidence on this question.

Large language models are available across a wide spectrum of sizes and deployment modes. Cloud-hosted frontier models offer strong extraction performance but transmit data to external servers. In banking contexts, where client data is subject to strict confidentiality requirements and data residency regulations, external data transmission may be impermissible. Smaller, locally deployable models can run entirely on-premises but typically exhibit lower accuracy on schema-conditioned extraction tasks. This thesis evaluates four models of different sizes and cost tiers in uniform-model configurations to characterise the accuracy, cost and latency trade-offs. Direct evaluation of mixed-model configurations, where individual agent roles are assigned different models, remains an open direction for future research.

1.4 Ethical and Regulatory Considerations

The deployment of an AI system for bank client onboarding is subject to ethical and regulatory constraints distinct from the purely technical challenges. This section identifies the most relevant constraints and documents how each is addressed within the scope of this thesis. The system is evaluated in a controlled sandbox environment; a production deployment would require a formal compliance assessment.

1.4.1 High-Risk Classification and Human Oversight

The EU AI Act classifies AI systems used in financial services and KYC/AML compliance as high-risk under Annex III [7]. High-risk systems are subject to obligations including human oversight, transparency and accuracy requirements. The present system addresses several of these by design: the preview-and-approval workflow (Section 5.7.12) implements a mandatory human oversight gate before any data is committed to Atfinity’s CMS and the evaluation framework provides documented accuracy measurements across multiple model configurations and case complexity tiers. The current evaluation does not include a systematic bias analysis across client profiles, which represents an open validation requirement for any production deployment.

1.4.2 Data Protection, Residency and Outsourcing Compliance

The system transmits client personal data to external AI provider APIs for processing. Under GDPR and the Swiss Federal Act on Data Protection (nLPD), this constitutes international data transfer to third-country processors [6]. In Switzerland, FINMA additionally classifies cloud-hosted LLM inference as an outsourced service, requiring a risk assessment and exit strategy before deployment at a regulated institution [17]. The LiteLLM abstraction layer (Section 5.7.3) directly supports a data-residency-compliant deployment by allowing all agent LLM assignments to be replaced with locally deployable model endpoints without code changes, eliminating external data transfer entirely where required.

1.4.3 Accountability and the Audit Trail

A compliance-critical onboarding system must provide a complete audit trail of all decisions. The present system supports this through two mechanisms: the preview-and-approval workflow records the Relationship Manager’s explicit confirmation before submission and the SchemaDB volume maintains a time-stamped archive of every accepted payload (Section 5.6.4). The system does not currently log the intermediate extraction outputs of individual Ontology Agents or the routing decisions of the Orchestrator in a persistent audit store, which represents a gap for any production deployment at a regulated institution.

1.5 Goals

The overarching goal of this thesis is to scale the single-agent prototype developed in the preceding study thesis into a multi-agent system capable of handling multi-party client onboarding scenarios. The study thesis established that MCP-based LLM integration with Atfinity’s CMS is technically feasible for simple, single-party cases. The present work extends this foundation by introducing a specialised multi-agent architecture in which each party type is handled by a dedicated agent, isolating the extraction context per entity and ensuring each party’s data is correctly assigned within a predefined payload structure submitted to Atfinity. The ultimate objective is to demonstrate that a multi-agent LLM setup of this kind is viable for production use at Atfinity, capable of handling the schema variability, multi-party complexity and runtime instance management that real-world onboarding cases require.

1.5.1 Product Objectives

The product must fulfill the following core objectives:

- Implement a modular multi-agent architecture in which specialised agents collaboratively handle the full onboarding workflow, with independently configurable LLM backends per agent role
- Accept natural language prompts describing multi-party client information and correctly extract and assign field data per entity, independent of how explicitly or completely the information is stated in the input
- Adapt to the currently deployed bank’s schema and process definitions at runtime, resolving the applicable workflow process and entity types for each case without relying on hard-coded field mappings
- Evaluate language models of varying sizes and cost tiers in uniform-model configurations, measuring trade-offs in extraction accuracy, runtime, token usage and cost across the multi-agent pipeline
- Automate test case generation from Atfinity’s validated onboarding cases, producing diverse scenarios across multiple conversational archetypes (see Section 6.11.1), multi-party configurations and edge cases to enable repeatable, statistically grounded evaluation
- Log and compare extraction quality metrics (precision, recall, F1 score and workflow completion rate), runtime, token usage and cost across all evaluated model configurations
- Production ready solution for Atfinity (optional)
- Prompting the user for input when required (optional)

1.5.2 Research Questions

The following research questions, as defined in the task description, guide the development and evaluation of the system:

1. **RQ1 (Accuracy & Robustness):** How does multi-agent decomposition affect extraction correctness and workflow reliability?
 - This question evaluates whether distributing extraction across specialised agents reduces role-misattribution errors, omissions and failed workflows. Success is measured using field-level precision, recall and F1 score. The single-agent designs from the study thesis serve as the qualitative reference point, demonstrating that the structural failure modes they exhibited on multi-party cases are resolved by the multi-agent decomposition.
2. **RQ2 (Efficiency):** How do LLM size and model family affect runtime, LLM call count and operational cost within the multi-agent architecture?
 - This question measures the end-to-end execution time from prompt submission to completed case creation, the total number of LLM invocations per onboarding run and the associated token cost. Results are compared across the four evaluated uniform-model configurations.
3. **RQ3 (Agent Specialisation):** Does the empirical evidence from uniform-model configurations support role-differentiated model assignment as a viable optimisation strategy?

- This question examines whether the accuracy and cost results observed across different uniform-model configurations provide indirect evidence that assigning smaller, cheaper models to lower-complexity agent roles while reserving larger models for extraction-intensive roles would yield a measurable advantage. A direct empirical evaluation of mixed-model configurations is not conducted within the scope of this work; the analysis draws theoretical inferences from the single-model results.
4. **RQ4 (Explainability, optional):** How can an Explainability Agent improve interpretability and trust in the system’s outputs?
- This optional question examines whether generating human-readable explanations for extracted field assignments increases Relationship Manager confidence in the system’s decisions and reduces the rate of manual corrections during the review-and-approval step.
5. **RQ5 (Architecture Necessity):** What structural constraints of multi-party bank client onboarding require a multi-agent design and is the operational cost justified against the manual alternative?
- This question identifies the structural constraints of multi-party bank client onboarding that make a multi-agent design a necessity rather than a performance optimisation, drawing on the study thesis as the single-agent reference point. It then evaluates whether the operational cost of the multi-agent pipeline is economically justified, not by comparing it against a hypothetical single-agent alternative, but against the manual onboarding process the system is designed to replace.

The answers to all research questions are presented and discussed in Section 7.7.

1.5.3 Research Value

An important aspect of this project is that the research questions carry value independent of whether a fully production-ready system is achieved. Answers to each research question provide concrete insights for Atfinity and the broader community of practitioners working on multi-agent LLM systems and MCP-based enterprise integrations. Understanding how multi-agent decomposition affects extraction correctness in schema-driven workflows, how LLM size and role specialisation affect cost and accuracy and what architectural patterns are required for reliable multi-party onboarding all contribute to the field. This holds regardless of whether all implementation goals are fully realised. The findings serve as a foundation for future iterations and production hardening.

1.6 Deliverables

The expected deliverables are:

- Product: Source code
- Comprehensive documentation of all aspects of the developed systems including answers to research questions
- Evaluation report with plots, tables and discussion of trade-offs

All deliverables except for the source code are part of this document. The complete source code and blog article are provided separately.

1.7 Abbreviations

- **ADR** - Architectural Decision Record
- **AI** - Artificial Intelligence
- **AML** - Anti-Money Laundering
- **AMLA** - Anti-Money Laundering Act (Geldwäschereigesetz, GwG)
- **API** - Application Programming Interface

- **BA** - Bachelor Thesis (Bachelorarbeit)
- **CI/CD** - Continuous Integration / Continuous Deployment
- **CMS** - Case Management System (Atfinity's platform for managing client onboarding cases)
- **FATF** - Financial Action Task Force
- **FINMA** - Swiss Financial Market Supervisory Authority
- **FR** - Functional Requirement
- **GDPR** - General Data Protection Regulation
- **GT** - Ground Truth
- **HTTP** - Hypertext Transfer Protocol
- **HTTPS** - Hypertext Transfer Protocol Secure
- **JSON** - JavaScript Object Notation
- **KYB** - Know Your Business
- **KYC** - Know Your Customer
- **LLM** - Large Language Model
- **MAS** - Multi-Agent System
- **MA** - Multi-Agent
- **MCP** - Model Context Protocol
- **MiFID** - Markets in Financial Instruments Directive
- **MVP** - Minimum Viable Product
- **NC** - New Case
- **NER** - Named Entity Recognition
- **NFR** - Non-Functional Requirement
- **NLP** - Natural Language Processing
- **PDA** - Process Determination Agent
- **PEP** - Politically Exposed Person
- **REST** - Representational State Transfer
- **RM** - Relationship Manager
- **RUP** - Rational Unified Process
- **SA** - Study Thesis (Studienarbeit)
- **TLS** - Transport Layer Security
- **UI** - User Interface
- **YAML** - YAML Ain't Markup Language

Part IV

Product Documentation

Chapter 2

Requirements

2.1 Use Case Overview

The use case diagram illustrates how the Relationship Manager interacts with the *MAS Preemptive Single-Shot* system to perform AI-driven, automated client onboarding and case management. Compared to the Study Thesis prototype, the Bachelor Thesis system exposes a substantially broader set of operations: party and role management, process discovery and a preview-and-approval workflow that allows the RM to verify and correct extracted data before any case is committed to Atfinity.

The system involves two actors:

- **Relationship Manager (RM):** The primary human actor. The RM interacts with the system exclusively through natural language via the LibreChat frontend, initiating case creation, reading and updating case data, managing parties and their roles, approving or correcting previews and querying available processes.
- **Multi-Agent System (MAS):** The system actor. The MAS participates in every use case because it is the execution layer that carries out whatever the RM requests. It coordinates a pipeline of specialised agents to interpret intent, extract structured data, assemble payloads and communicate with Atfinity's backend via the MCP layer (Model Context Protocol, introduced in Chapter 3). (Detailed Core Architecture)

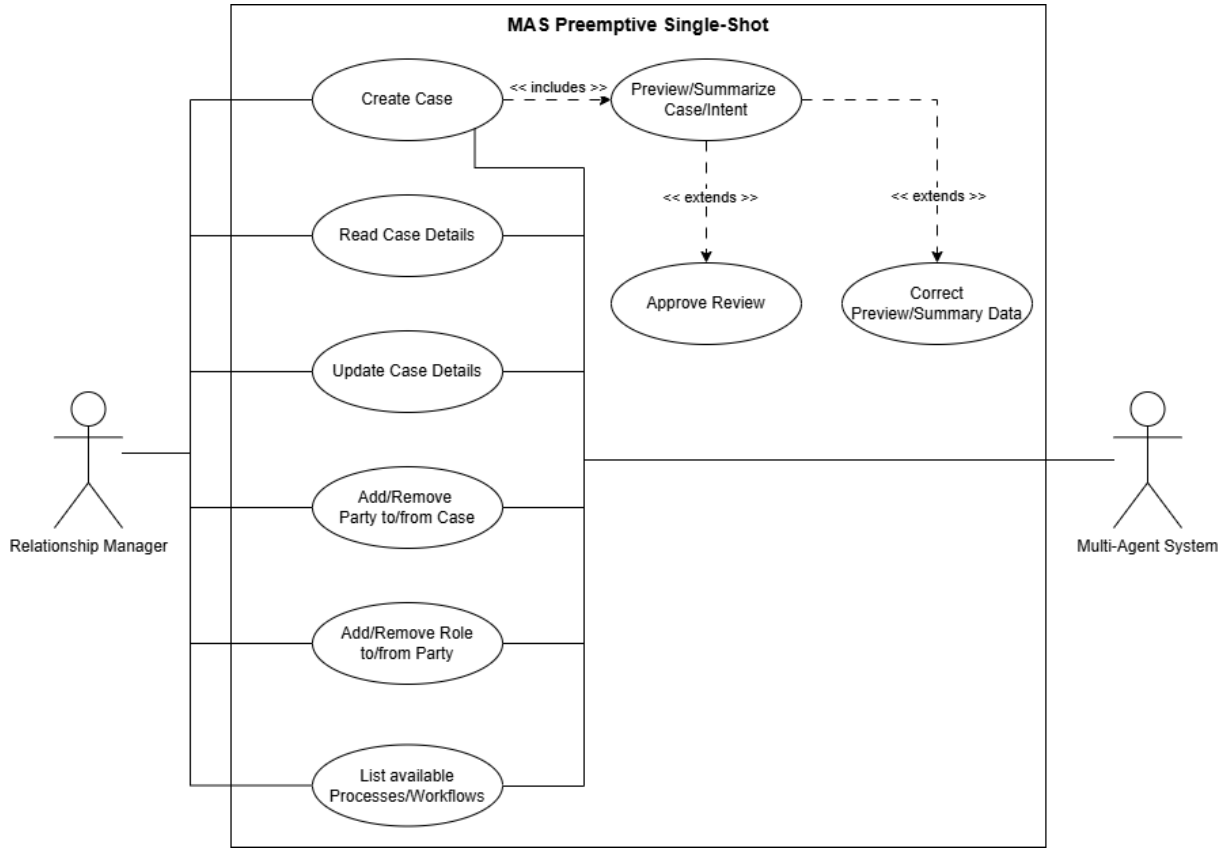


Figure 2.1: Use Case Diagram: Multi-agent Preemptive Single-Shot

The diagram captures six primary use cases. **Create Case** includes **Preview/Summarize Case/Intent**, meaning a preview is always generated before submission. The preview step itself extends to two conditional branches: **Approve Review**, triggered when the RM confirms the preview is correct and **Correct Preview/Summary Data**, triggered when the RM identifies errors and provides updated information. **Read Case Details**, **Update Case Details**, **Add/Remove Party to/from Case**, **Add/Remove Role to/from Party** and **List available Processes/Workflows** are independent use cases the RM can invoke at any time on an existing case.

2.2 User Stories

User stories capture the needs and workflows of Relationship Managers using the system. User Stories 1, 3, 4 and 5 carry over from the Study Thesis; User Stories 2, 6, 7 and 8 reflect capabilities introduced in the Bachelor Thesis.

2.2.1 User Story 1: Create Automatic Client Onboarding

As a Relationship Manager, I want to conversationally input client information using natural language, which will automatically populate and fill out the onboarding case, so that I can avoid manually filling out lengthy and repetitive forms.

2.2.2 User Story 2: Preview and Approve Case Before Submission

As a Relationship Manager, I want the system to present a structured summary of the extracted client data and the identified workflow before any case is created, so that I can verify the interpretation is correct and correct any errors before they are committed to Atfinity.

2.2.3 User Story 3: Ask for Still Open Fields

As a Relationship Manager, I want the system to understand which fields of an existing case are currently required but not yet filled out, so that I know exactly what information I still need to provide.

2.2.4 User Story 4: Update Existing Case

As a Relationship Manager, I want to update one or more fields of an existing case using natural language, without needing to know field identifiers or navigate the system manually.

2.2.5 User Story 5: Get Relevant Information and Ask Questions About a Case

As a Relationship Manager, I want to ask questions about a specific case so that I can retrieve relevant information without navigating the system manually.

2.2.6 User Story 6: Add or Remove a Party from an Existing Case

As a Relationship Manager, I want to add a new party (e.g., a person or legal entity) to an existing case or remove an existing party from it, so that the case accurately reflects the current set of involved persons and entities.

2.2.7 User Story 7: Manage the Roles of a Party

As a Relationship Manager, I want to assign, change or revoke roles for an existing party within a case, so that the case reflects each person's or entity's correct legal or functional relationship to the account.

2.2.8 User Story 8: Discover Available Processes and Workflows

As a Relationship Manager, I want to ask the system which onboarding process or workflow applies to a given scenario, so that I can confirm the correct process is used before initiating a case.

2.3 Functional Requirements

Functional requirements specify what the system must do to fulfil the user stories described above.

2.3.1 FR1: Retrieve Case Information and Answer Questions

The system must fetch the current state of a case from Atfinity's CMS, including all instance data (parties, roles, field values) and metadata. The multi-agent system must synthesise this information into a human-readable summary and answer the RM's questions via the frontend. The system must always fetch live data directly from Atfinity rather than relying on any cached or previously retrieved state, ensuring the RM receives an accurate representation of the case at the moment of the query. (Supports User Stories 3, 5)

2.3.2 FR2: Extract Structured Data from Free-Text Input

The system must process free-text input provided by a Relationship Manager and automatically extract structured field values for all involved parties. Extraction must correctly handle variations in phrasing, numerical formats and expressions of uncertainty (e.g., "approximately CHF 100,000"). (Supports User Stories 1, 4)

2.3.3 FR3: Adapt to Atfinity's Incremental Field Architecture

The system must adapt to Atfinity's incremental case structure, where field visibility depends on previously submitted information and must assemble payloads in the exact format Atfinity's API requires. (Supports User Story 1)

2.3.4 FR4: Detect and Route User Intent

The system must classify incoming user messages and route them to the appropriate tool. The Orchestrator (the central coordination component, described in Chapter 5) must distinguish between read operations (questions about case state), write operations (data input, field updates, party or role changes) and discovery operations (queries about available processes). For write operations, the Process Agent must identify the correct Atfinity process from the user's prompt with a minimum confidence threshold of 0.6 before proceeding. If no process can be matched above this threshold, the system must reject the request and inform the RM. (Supports all User Stories)

2.3.5 FR5: Expose All Operations as MCP Tools

The system must provide all core operations as MCP tools accessible through the LibreChat frontend. Each tool must accept natural-language input and return structured JSON responses conforming to the MCP specification. The following tools must be available:

- **create_client_onboarding**: runs the full MAS pipeline to create a new case
- **preview_onboarding**: generates a pre-submission approval preview for the RM
- **read_case**: fetches the live state of a case including all instances and roles
- **update_case_field**: updates one or more fields in an existing case
- **add_case_instance**: adds a new party instance to a case
- **delete_case_instance**: removes a party instance from a case
- **add_instance_role**: assigns a role to an existing instance
- **remove_instance_role**: revokes a role from an existing instance
- **set_instance_roles**: replaces the full role set of an existing instance by computing the delta between the current and desired roles
- **list_onboarding_processes**: identifies the most relevant Atfinity process for a given scenario

(Supports all User Stories)

2.3.6 FR6: Preview and Approval Workflow

The system must provide a preemptive approval step before any case is created. The **preview_onboarding** tool must execute the full extraction pipeline and return the organised client data, the matched process, the identified business partner type and the list of entities that will be created, allowing the Relationship Manager to verify and correct the results before any case is committed to Atfinity. If the RM provides corrections, the system must minimise redundant computation by reusing cached results from the previous preview where the input has not changed. When the RM approves, **create_client_onboarding** must reuse all cached results from the approved preview so that no redundant LLM calls are made. (Supports User Story 2)

2.3.7 FR7: Party Instance Management

The system must allow the RM to add new party instances (e.g., Person, Entity) to an existing case and to remove existing instances. For addition, the system must resolve the correct ontology type and any role assignments from the natural-language prompt before calling Atfinity's instance creation endpoint. For removal, the system must attempt a hard delete first; if Atfinity does not permit hard deletion, the system must fall back to stripping all roles from the instance, rendering it inactive. (Supports User Story 6)

2.3.8 FR8: Role Management

The system must allow the RM to add, remove or replace roles on any existing instance within a case, using natural language to identify the target instance and the intended role operation. The system must resolve instance identities by fetching the live case state and must validate role keys against the available ontology catalogue before performing any role operation. (Supports User Story 7)

2.3.9 FR9: Process Discovery

The system must allow the RM to query which Atfinity process or workflow applies to a given scenario without initiating a case. The Process Agent must match the prompt against its in-memory process context and return the best-matching process with its identifier, key, display name, confidence score and reasoning. If no process can be matched with sufficient confidence, the system must inform the RM and ask for a more specific description. (Supports User Story 8)

2.3.10 FR10: Autonomous Self-Healing on Submission Errors

When Atfinity rejects a submitted payload, the system must autonomously diagnose and correct the errors without requiring RM intervention and retry submission up to five times before surfacing a failure to the Relationship Manager. The correction strategy must vary by error type, distinguishing at minimum between structural format errors, semantic value errors and ontology-level structural errors. (Supports User Story 1)

2.4 Non-Functional Requirements

Non-functional requirements specify the quality criteria against which the system's operation is evaluated.

2.4.1 NFR-1: Performance

The end-to-end pipeline, from the RM's natural-language prompt to a successfully created case in Atfinity, must complete within the acceptance threshold defined below. This includes all agent calls (Prompt Organizer, Process Agent, Ontology Agents in parallel), payload assembly and at least one submission attempt to Atfinity. The threshold is set higher than in the Study Thesis to account for the increased complexity of multi-agent coordination, parallel extraction across large ontologies and the self-healing retry mechanism.

- Acceptance: ≤ 120 sec

Verification: Benchmark the architecture under controlled test conditions using realistic onboarding prompts. Measure end-to-end wall-clock time for at least three representative test cases, each targeting a different process and party configuration. Retry-induced latency is included in the measurement.

2.4.2 NFR-2: Precision

Precision ensures that extracted field values are correct and minimises the risk of incorrect data being submitted to Atfinity. In a KYC/AML banking context, a low precision threshold is not acceptable: an incorrectly extracted field value may propagate silently into the case record, creating a compliance risk that the self-healing loop cannot detect since the value is structurally valid. The ratio of correctly extracted fields among all fields extracted by the system must meet the following target:

- Acceptance: ≥ 0.90 (90%)

Verification: Evaluate extracted fields against human-validated gold-standard reference data derived from Atfinity's sample onboarding cases.

2.4.3 NFR-3: Recall

Recall ensures that all relevant information present in the RM's input is captured and that no required field is overlooked during extraction. The ratio of correctly extracted fields among all fields that should have been extracted must meet the following target:

- Acceptance: ≥ 0.90 (90%)

Verification: Evaluate extracted fields against human-validated gold-standard reference data derived from Atfinity's sample onboarding cases.

2.4.4 NFR-4: F1-Score

The F1-score balances correctness (precision) and completeness (recall), providing the primary aggregate measure of extraction quality. It ensures the system neither sacrifices accuracy for coverage nor achieves high accuracy at the cost of missing information. The F1-score, defined as the harmonic mean of precision and recall, must achieve the following target:

- Acceptance: ≥ 0.85 (85%)

Verification: Evaluate extracted fields against human-validated gold-standard reference data derived from Atfinity’s sample onboarding cases.

2.4.5 NFR-5: Efficiency

Efficiency quantifies the resource consumption and operational cost of the multi-agent pipeline per onboarding case. Given that multi-agent architectures issue multiple parallel LLM calls per case, understanding the cost implications across different model configurations is an explicit evaluation objective of this thesis. The following metrics are recorded per test case execution:

- Total number of LLM calls issued across all agents
- Total token usage (prompt tokens and completion tokens separately)
- Estimated API cost in USD, computed per model using per-token pricing at the time of evaluation

These metrics are compared across the four evaluated uniform-model configurations to characterise the efficiency and cost trade-offs addressed by Research Questions RQ2 and RQ3.

Verification: Instrument the pipeline to log prompt token usage, completion token usage and USD cost per LiteLLM model string per test case, aggregated across all agent calls sharing the same model. Aggregate run-level totals per test case and compare across all evaluated uniform-model configurations.

2.4.6 NFR-6: Usability

The system must handle the RM’s free-text prompts in natural language without requiring knowledge of Atfinity’s internal field identifiers, ontology keys, instance IDs or process numbers. All inputs from the RM must be expressed in human-readable terms; all outputs must be presented as plain-language summaries.

Verification: A Relationship Manager can enter a free-text prompt containing client information and receive a correctly populated case confirmation without providing any technical identifiers or system-internal terms.

Chapter 3

Model Context Protocol (MCP)

This chapter describes the Model Context Protocol, the communication standard that connects the multi-agent system to the LibreChat frontend and to Atfinity’s backend. Section by section, it covers what MCP is, how it works, what capabilities it exposes and why it was chosen as the integration layer for this project.

3.1 Definition and Purpose

The Model Context Protocol is a standardised, open-source communication protocol that enables large language models to interact with external systems, data sources and tools in a uniform and interoperable way [12]. Before MCP, integrating an LLM with an external system required custom, tightly coupled code for every combination of model and backend. Each integration was application-specific: swapping the LLM or replacing a backend service demanded changes throughout the stack and there was no shared mechanism by which a model could discover what operations were available to it [12].

The concept of giving an LLM access to external operations is not new. OpenAI introduced function calling in 2023 as a way for GPT models to request the execution of developer-defined functions by generating structured JSON arguments [15]. MCP generalises and standardises this idea: rather than a model-specific API feature, it defines a protocol layer that any model and any backend can implement independently. The result is a universal interface, analogous to a USB-C hub, through which any MCP-compatible client can interact with any MCP-compatible server without custom adapters [12].

Figure 3.1 illustrates this universal interface concept.

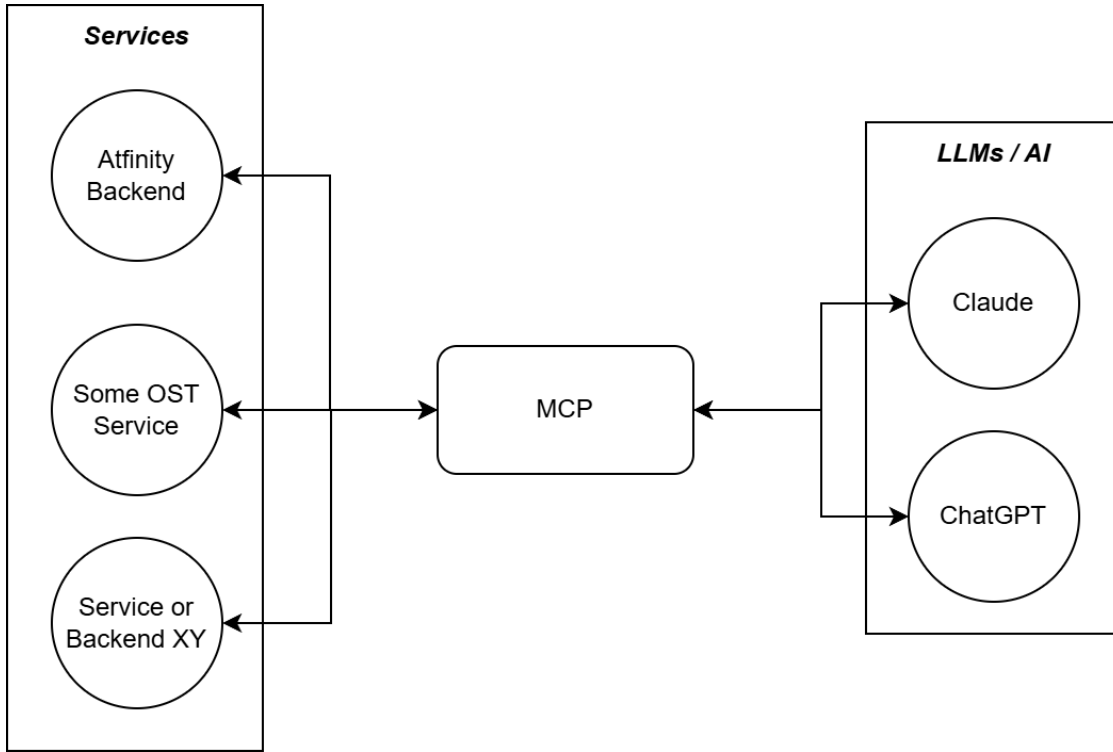


Figure 3.1: MCP as a universal interface layer between LLM clients and backend systems [12]

3.2 Architecture and Communication

MCP operates on a client-server model [12]. An **MCP Server** exposes capabilities, tools, resources and prompts through the protocol. It implements the domain-specific logic (for example, querying a banking CMS or executing a field extraction) and returns structured JSON responses to callers. An **MCP Client**, typically embedded in an AI application, connects to one or more servers, retrieves their capability manifests and mediates between the LLM and the server during execution.

Communication follows a defined message format. At initialisation the client performs a capability discovery handshake, after which the descriptions of all available tools are inserted into the LLM’s context window [12]. From that point, the LLM can reason about which tool is appropriate for the user’s request, generate a structured invocation and receive the result, all within its normal inference loop. This tool-use loop can iterate: the model may call multiple tools sequentially, incorporating each result before deciding on the next action [15].

3.3 Exposed Capabilities

MCP servers can expose three categories of capability [12]:

- **Tools:** Executable operations with well-defined input parameters, output schemas and descriptions. The LLM invokes a tool by generating the correct arguments; the MCP client validates the call and forwards it to the server. Tools are the primary mechanism for write operations and computations.
- **Resources:** Read-only, referenceable data that the LLM can retrieve and incorporate into its reasoning, for example documents, configuration files or database snapshots. Unlike tools, resources do not trigger side effects.
- **Prompts:** Pre-defined instruction templates that a server can expose to standardise LLM behaviour across applications, such as task-specific extraction instructions or compliance-oriented question templates.

3.4 Relevance to This Project

In this project, MCP serves as the interface layer between the Relationship Manager and the full multi-agent pipeline. All system capabilities are exposed as ten MCP tools, covering the complete range of operations described in Chapter 2: case creation, preemptive preview and approval, case reading, field updates, party instance management, role management and process discovery. Each tool accepts a natural-language prompt and returns a structured JSON response conforming to the MCP specification, making the interface uniform regardless of the complexity of the underlying agent pipeline.

This architecture provides three concrete benefits. First, the MCP layer fully decouples the LibreChat frontend from the multi-agent orchestration logic and from Atfinity’s backend. The frontend is aware only of the tool interface and has no knowledge of how many agents are involved, which LLM each uses or how Atfinity’s API is structured. Second, because each agent in the pipeline communicates via its own internal interface, different models can back different agents, a more capable model for extraction and a smaller model for routing, without any change to the MCP tool signatures [12]. This is directly relevant to Research Questions RQ2 and RQ3 (Section 1.5), which require comparing LLM configurations across agent roles. Third, the tool-based model is inherently extensible: new capabilities can be added to the server without modifying existing tools or the client.

Chapter 4

Multi-Agent Systems

This chapter provides the theoretical foundation for the multi-agent design used in this project. It defines what a multi-agent system is and why the paradigm is well-suited to complex LLM-based workflows. It then describes the agent roles and specialisation principles, the control topologies and execution patterns that govern how agents collaborate and the properties of LLM-based agents. The final section connects these concepts directly to the design decisions of the present work.

4.1 Definition and Motivation

A multi-agent system (MAS) is a computational system composed of multiple autonomous agents, each of which perceives its environment, maintains some form of internal state and acts to achieve a defined objective [18]. Crucially, no single agent possesses complete knowledge of the overall task. Each agent is responsible for a bounded sub-problem and the system-level goal emerges from their coordinated interaction. This principle of decomposition is the central motivation for the MAS paradigm: complex problems that exceed the capacity of any single reasoner are tractable when divided into well-scoped, independently solvable parts [18].

In the context of LLM-based systems, the multi-agent paradigm has gained significant traction as a practical response to the limitations of single-model pipelines [19]. A single LLM handling a complex, multi-step task faces three structural constraints. First, context window limits bound the amount of information the model can process in one pass; large schemas or multi-party inputs may not fit within a single context. Second, a prompt that combines multiple responsibilities, extraction, intent classification, validation and routing, increases the likelihood of the model conflating tasks or accumulating errors across steps. Third, a single model cannot be simultaneously optimised for tasks with fundamentally different requirements, such as low-latency classification and high-accuracy structured extraction from domain-specific schemas [19]. Multi-agent decomposition addresses all three by assigning each concern to a dedicated agent with a focused prompt, appropriate model and scoped context.

4.2 Agent Roles and Specialisation

Agent specialisation is the defining design principle of a MAS: an agent that performs exactly one task is easier to prompt, easier to evaluate and more reliable than a general-purpose agent [19]. The role assigned to an agent determines the class of inputs it accepts, the reasoning it performs and the outputs it produces. In LLM-based architectures, the following roles appear recurrently [19, 18]:

- **Orchestrator:** Coordinates the overall pipeline. The orchestrator receives the top-level request, decides which sub-agents to invoke, in what order and how to assemble their outputs. It holds no domain knowledge itself; its sole responsibility is flow control and result aggregation.
- **Extraction Agent:** Identifies and extracts structured field values from unstructured natural-language input. Extraction agents are scoped to a specific data domain or ontology schema so that their prompt context remains focused and the risk of schema confusion is minimised.

- **Classification / Routing Agent:** Analyses the intent of an incoming request and determines which downstream process or agent should handle it. A routing agent prioritises speed and categorical reliability over exhaustive reasoning.
- **Review / Validation Agent:** Performs quality control on the outputs of other agents. Rather than producing new data, it verifies consistency and correctness and triggers re-extraction when it detects contradictions or missing values [19].
- **Knowledge / Cache Agent:** Stores previously computed mappings or results so that downstream agents can retrieve them without repeating expensive LLM calls, improving both latency and cost efficiency.

4.3 Control Topologies

Before considering how agents sequence their work, it is important to understand how control and decision-making authority are distributed across the system. IBM identifies three fundamental control topologies for multi-agent architectures [10]:

4.3.1 Centralised

In a centralised topology, a single coordinator agent holds all decision-making authority [10, 13]. This configuration is also referred to as the Orchestrator-Executor pattern: the central node assigns tasks to specialised worker agents and assembles their results [13]. All other agents report to and receive instructions from this central node. The coordinator has global visibility over the system state and is solely responsible for task allocation, sequencing and result aggregation. This topology is straightforward to reason about and produces consistent, predictable behaviour. Its primary limitation is that the coordinator represents a single point of failure and a potential throughput bottleneck as the number of sub-agents grows.

4.3.2 Decentralised

In a decentralised topology, agents interact as peers without any single controlling node [10]. Each agent makes its own decisions based on local information and coordinates with others through direct message passing. Decentralised systems are more resilient, with no single failure halting the entire pipeline and scale more naturally to large agent populations. The trade-off is coordination complexity: without global oversight, ensuring all agents converge on a coherent outcome requires explicit negotiation protocols [18].

4.3.3 Hierarchical

A hierarchical topology organises agents into levels of authority, with higher-level agents coordinating those below [10, 9]. Unlike a purely centralised system, coordination work is distributed across levels: each node manages only its immediate sub-agents, reducing the load on any single coordinator. This topology balances the predictability of central oversight with the scalability of distributed execution and is the most widely adopted pattern in production LLM-based multi-agent systems [9].

4.4 Execution Patterns

Distinct from control topology, which governs authority distribution, execution patterns describe how agents sequence and parallelise their work at runtime. Three patterns are foundational [19, 18]:

4.4.1 Sequential

Agents are arranged in a fixed pipeline: each agent receives the output of its predecessor and passes its own result forward. Sequential execution is straightforward to reason about and debug, but total latency equals the sum of all individual agent latencies and an error in any stage propagates to all subsequent stages.

4.4.2 Parallel

Multiple agents operate simultaneously on independent sub-problems. A coordinating layer fans out work and aggregates results once all agents have finished, a configuration referred to as Parallel Fan-Out/Fan-In in agent architecture literature [13]. Parallel execution is particularly effective when the same task type must be applied to multiple independent inputs, for example extracting fields from several distinct parties simultaneously. The latency of the parallel group is bounded by the slowest individual agent, not the sum [19].

4.4.3 Hybrid

Hybrid architectures combine sequential and parallel execution: sequential ordering governs stages where one step’s output is a strict prerequisite for the next, while parallel execution is applied within stages where independence permits it [19]. This pattern is the most expressive and is well-suited to workflows that contain both ordering constraints and opportunities for concurrent execution.

4.5 LLM-Based Agents

A classical software agent operates by rules: conditions map to actions deterministically. An LLM-based agent replaces this fixed logic with a language model as the reasoning core [20]. Given a system prompt defining its role, a set of available tools and the current context, the model determines dynamically what action to take, invoking a tool, producing a structured output or requesting clarification. This architecture was formalised in the ReAct paradigm, which interleaves reasoning traces with concrete actions, allowing the model to ground its decisions in intermediate observations rather than relying solely on its parametric knowledge [20].

LLM-based agents offer two properties that rule-based agents cannot match. They generalise across input variations without requiring explicit enumeration of cases: an extraction agent will correctly map “approximately CHF 100,000” to a numeric field without a hand-crafted rule for that phrasing. And they are steerable through prompt engineering: their behaviour can be refined by modifying the system prompt or tool descriptions, without changing application code [19].

The critical reliability mechanism for LLM-based agents is *scope control* [19]. A prompt that attempts to cover too many responsibilities forces the model to maintain an excessively large reasoning context, increasing the probability of errors, hallucinations or constraint violations. Assigning each agent precisely the information and instructions it requires and no more, is the primary design lever for building reliable LLM-based pipelines.

4.6 Relevance to This Project

The three limitations of the single-agent prototype that motivate the multi-agent design of this work, conditional field exposure, LLM size and cost trade-offs and evaluation coverage, are established in Section 1.2. Each maps directly onto a design principle from this chapter: agent decomposition addresses the context boundary problem by assigning each entity type its own scoped extraction agent; model-per-role assignment addresses the cost and accuracy trade-off by allowing different agents to use models suited to their specific reasoning requirements; and the modular pipeline structure enables per-stage logging of inputs, outputs and latency for automated evaluation.

The architecture that applies these principles in practice, the *Multi-agent Preemptive Single-Shot*, uses the hierarchical control topology and the hybrid execution pattern described in the preceding sections. Its structural details are described in full in Chapter 5.

Chapter 5

Design / Architecture

This chapter documents the architectural development and the final design of the multi-agent system. The Preamble introduces the C4 model notation used throughout. Section 5.2 presents the system context diagram. Section 5.3 traces the architectural context, briefly summarising the study thesis baseline, the increased complexity of the present work and the foundational decisions carried forward. Sections 5.4 and 5.5 examine the Iterative and Hybrid approaches from the study thesis and explain why neither could be applied in the multi-agent, multi-party setting of the present work. Section 5.6 documents the Multi-agent Preemptive Single-Shot architecture in full, covering its deployment environment, containerised topology, a detailed description of each system component, a dedicated explanation of how schema and prompt context is scoped per agent (Section 5.6.5), the end-to-end case creation pipeline with its six-step message sequence and a quality attributes analysis showing how the design decisions address the system’s non-functional requirements. Section 5.7 presents the Architectural Decision Records, documenting the key design choices made during the bachelor thesis phase and the rationale behind each decision.

5.1 Preamble

For documenting the architecture diagrams, the C4 model was chosen as the primary framework. The C4 model provides a hierarchical approach to visualising software architecture, decomposing the system into four progressively detailed levels: System Context, Container, Component and Code. This layered structure allows each level to address a different audience, from non-technical stakeholders at the context level to implementers at the component level. This chapter covers the System Context and Container levels, as these are the levels most relevant to communicating the architectural structure and runtime dependencies of the system. The Component and Code levels are omitted, as the implementation chapter provides the necessary technical depth at those granularities. [5]

5.2 System Context Diagram

Figure 5.1 provides the highest-level view of the system, situating it among its users and the external systems it depends on. The Relationship Manager interacts with the system exclusively through a web-based frontend. The system communicates with Atfinity’s CMS to retrieve schema definitions, submit case payloads and manage party instances. All LLM inference calls are routed to external provider APIs, which operate outside the system boundary and are not shown as separate components at this level.

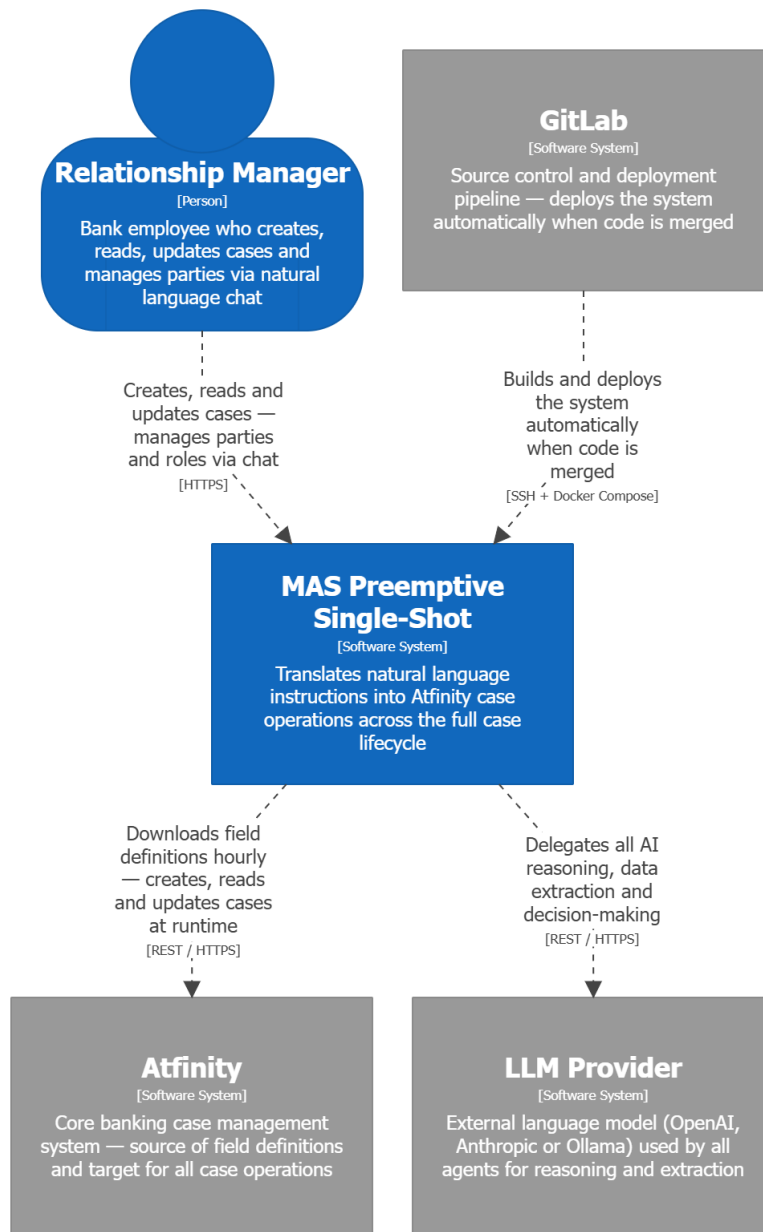


Figure 5.1: System Context Diagram

5.3 Architectural Overview

The following subsections establish the architectural context for the present work. The study thesis baseline is summarised first, including the foundational decisions carried forward into the present work, followed by an explanation of the increased complexity that the multi-agent, multi-party problem introduces.

5.3.1 Study Thesis Baseline

As described in Section 1.2, the study thesis evaluated three single-agent architectures, Iterative, Hybrid and Preemptive and confirmed that LLM-driven extraction with Atfinity’s CMS via MCP is technically viable for simple, single-party onboarding scenarios. A set of foundational infrastructure decisions from that work are carried forward here unchanged.

LibreChat is retained as the frontend and MCP client, given its native Anthropic API support, MCP-aware design and full containerisability. Docker Compose is retained as the deployment and orchestration tool and all components are containerised from the outset, avoiding the local dependency and portability issues that burdened the earlier iterations. The external-network-based container communication pattern and the shared Docker volume approach for schema data are both preserved and extended to accommodate the more complex multi-agent topology. These decisions are not revisited here; they carry forward from the study thesis on the basis of validated findings.

5.3.2 Increased Complexity in the Bachelor Thesis

Grounding the system in a real-world bank deployment introduces a scale of complexity that goes beyond what the study thesis configurations exposed. For the bachelor thesis, a new Atfinity sandbox tenant was provisioned as a representative copy of the Bank of Luxembourg’s configuration. This tenant exposes 23 ontologies, 30 processes and approximately 25 000 fields across the entire schema. Passing this volume of schema data to an LLM in a single context window is not feasible. It exceeds practical context limits for reliable extraction and exposing the full field set in a single prompt dramatically increases the risk of hallucination. A model reasoning over 25 000 fields simultaneously may select plausible-looking but incorrect field keys, confuse structurally similar fields across ontologies or produce values that match the field name semantically but violate the permitted enumeration constraints. The result is degraded extraction precision and inflated processing times that scale with the size of the schema rather than with the complexity of the individual case.

All three architectural strategies from the study thesis were re-evaluated against these requirements. The Iterative approach and the Hybrid approach both proved infeasible for the multi-agent, multi-party setting, for reasons detailed in Sections 5.4 and 5.5 respectively. The Preemptive approach, extended with a predefined-object payload structure and a multi-agent decomposition, was the only design that satisfied the correctness, scalability and instance-tracking requirements of the present work. This architecture, designated the *Multi-agent Preemptive Single-Shot*, is described in detail in Section 5.6.

5.4 The Iterative Approach: Not Feasible in the Bachelor Thesis Context

The Iterative approach, as implemented in architecture v5 of the study thesis, operates as a closed feedback loop. After a blank case is created in Atfinity, the system reads the fields Atfinity has exposed at that point, extracts values for them via an LLM call, submits those values and then re-queries Atfinity for any newly exposed fields that the previous submission unlocked. This cycle continues until Atfinity exposes no new fields, at which point the case is considered complete. While this design was effective for simple, single-party onboarding cases, it is structurally incompatible with the multi-party scenario of the present work.

The fundamental problem is instance identifier tracking, introduced as the third technical obstacle in Section 1.2. In Atfinity, every field in a case is not identified by a flat field key alone; it is identified by a composite key of the form `<instanceId>-<fieldId>`. For a case with five natural persons, each person occupies a separate instance (see Section 1.1.1) and submitting the correct value to the correct

field requires knowing the exact instance identifier Atfinity has assigned to each person. These identifiers are not known before case creation; they are generated by Atfinity at the moment the case is created and are only discoverable by reading the case state afterwards.

A further complication arises from how Atfinity enforces compliance rules, also described in Section 1.2. When a specific business partner type is selected during onboarding, Atfinity does not wait for the user to define how many persons are involved. Instead, it automatically generates a set of party instances and assigns predefined roles to them based on the compliance requirements of that onboarding type. For example, an onboarding type that mandates the presence of a beneficial owner, an authorised signatory and a controlling person will cause Atfinity to create three instances immediately upon type selection, each pre-assigned its required role. The system therefore receives a set of auto-generated instances whose count is determined by the compliance rule, not by the number of persons described in the Relationship Manager’s prompt.

This creates an unresolvable ambiguity for the Iterative approach. The system sees, for example, five instances in the case but cannot determine from the case state alone which instance should receive the data of which person from the original prompt. A naive heuristic of mapping instances to persons by order of appearance is unreliable because Atfinity permits multiple roles to be assigned to a single instance. Two compliance-required roles may be satisfied by one person, meaning the number of instances in the case does not correspond to the number of individuals being onboarded. Any attempt to overwrite instance assignments or re-map roles after the fact risks violating the case’s compliance state and results in non-deterministic field attribution when persons share identical schemas. The Iterative approach provides no mechanism to resolve this ambiguity reliably and its sequential update cycle makes the problem worse with each additional party.

5.5 The Hybrid Approach: Not Feasible in the Bachelor Thesis Context

The Hybrid approach, as implemented in architecture v6 of the study thesis, combines an upfront preemptive field mapping with an iterative LLM fallback for fields that the initial mapping did not resolve. This design reduced the total number of LLM calls compared to pure iteration by covering the majority of fields in a single pass, reserving the iterative cycle only for fields exposed conditionally by Atfinity after the initial submission. Despite this efficiency improvement, the Hybrid approach is disqualified from the present work for the same structural reason as the Iterative approach. Its iterative fallback component is subject to the identical instance identifier tracking problem.

The preemptive mapping step is executed before the case is created in Atfinity and therefore before any instance identifiers are known. For a single-party case this is not an issue, since there is only one instance and all fields belong to it unambiguously. For a multi-party case, however, the preemptive mapping must assign each extracted field value to a specific instance. Because instance identifiers are generated by Atfinity at case creation time and depend on the compliance rules of the selected business partner type, they cannot be anticipated during the preemptive step. The preemptive mapping can at best produce a set of field-value pairs without reliable instance scoping, leaving the instance assignment problem entirely to the fallback iteration cycle.

The iterative fallback inherits the full problem described in Section 5.4: once the case exists and Atfinity has generated its compliance-driven instances, the system still cannot reliably determine which auto-generated instance corresponds to which person from the original prompt and the role-consolidation behaviour of Atfinity means the instance count does not map predictably to the number of individuals being onboarded. The Hybrid approach therefore does not escape the instance-tracking problem; it only defers it to the fallback phase. For multi-party cases, this deferral provides no benefit and the approach is disqualified on the same grounds.

5.6 Multi-agent Preemptive Single-Shot Architecture

This section describes the architecture selected for the present work. The architectural overview covers the fundamental design decision and the properties it enables. The deployment environment subsection documents the infrastructure on which the system runs. The container diagram presents the full deployment topology. The components subsection then provides a detailed description of each service, volume and external dependency in the system.

5.6.1 Architectural Overview

The Multi-agent Preemptive Single-Shot architecture departs from all study thesis designs in one fundamental respect: the entire case payload is assembled in a single coordinated pipeline before any call is made to Atfinity’s case creation endpoint. There is no incremental field submission, no iterative update cycle and no dependence on instance identifiers that only become available after case creation. Instead, the system assembles a complete `predefined_objects` payload that Atfinity accepts in a single bulk POST request, creating the case and all its party instances atomically in one operation. This eliminates the instance-tracking problem that disqualified the other two approaches.

This design enables the multi-agent decomposition that is central to the thesis. Because payload assembly is fully decoupled from case creation, multiple specialised Ontology Agents can extract fields in parallel, each operating on a focused sub-prompt that contains only the data relevant to its assigned entity. The Orchestrator collects all extraction results, assembles the master payload and submits it in a single atomic operation. An autonomous self-healing retry loop within the Orchestrator handles validation errors returned by Atfinity without requiring the Relationship Manager to re-enter data.

5.6.2 Deployment Environment

The entire system is deployed on a virtual machine provided by the Ostschweizer Fachhochschule (OST). All components run as Docker containers orchestrated by Docker Compose. The containers share a single Docker network, designated `onboarding-network`, through which all inter-service communication takes place. The OST VM provides a stable, persistently reachable environment for the LibreChat frontend and all agent microservices. The Atfinity CMS instance used for integration is deployed and operated separately by Atfinity AG and is accessed by the system exclusively over the network via Atfinity’s REST API. No components of the system are installed locally on any developer machine during evaluation; the entire stack runs on the OST VM.

The OST VM was selected as the exclusive deployment target to ensure that all evaluation results reported in Chapter 7 are measured under identical, controlled infrastructure conditions. Deploying the system on developer workstations would introduce variability in CPU performance, memory availability and network latency that would make latency and throughput measurements non-comparable across evaluation runs. By fixing the hardware, operating system and network path to Atfinity’s sandbox tenant, the benchmark results are reproducible and reflect the behaviour of the system under a stable, representative deployment environment. The rationale for this decision is formalised in Section 5.7.1.

The system runs in two separate environments on the same OST VM: a production environment mapped to the `main` branch and a development environment mapped to the `dev` branch. Both environments share the same Docker network and infrastructure services, MongoDB and the Caddy reverse proxy for HTTPS termination, but run independent instances of the multi-agent pipeline with separate configuration and environment variables. This separation allows new changes to be tested in full integration against the live Atfinity sandbox tenant before they are promoted to the production environment, without affecting the production deployment or its evaluation data.

All deployments are automated through a GitLab CI/CD pipeline that defines four sequential stages: `infrastructure`, `build`, `evaluate` and `deploy`. Standard deployment jobs trigger on pushes to the `dev` branch and on merges to `main`; evaluation runs and Sync Container builds are additionally gated on Git tags, which decouple those operations from routine code pushes. The infrastructure stage deploys MongoDB and Caddy and is change-triggered, executing only when files under the `infrastructure/` path, the pipeline definition itself or environment files have been modified. The build stage compiles three static Docker images on branch pushes, the Orchestrator, the Process Agent and the Prompt Organizer

Agent and pushes them to the GitLab Container Registry; the Sync Container image is built separately on tag events to isolate full schema synchronisation runs from ordinary deployments. Tags are namespaced: production jobs respond to tags that do not carry a `dev-` prefix, while development jobs respond to tags that do, ensuring that a tagged evaluation or sync run targets the correct environment. The evaluate stage is tag-triggered: it builds and executes a dedicated evaluation-runner container inside the shared Docker network and saves the resulting `results.json` as a CI artefact, making benchmark data reproducible and version-traceable per tagged release. The deploy stage transfers deployment artefacts to the OST VM via SSH and `rsync`, restarts the core service containers via Docker Compose and triggers a deploy-only pass of the Sync Container to provision the current set of Ontology Agent containers without executing a full schema resynchronisation against Atfinity. Environment-specific secrets are managed through GitLab Secure Files, downloaded by the CI runner at job time and never stored in the repository. A code push to `dev` or a merge to `main` is the only manual action required to deploy a new version of the system to the corresponding environment; tagged releases additionally trigger the evaluation and Sync Container deployment stages.

5.6.3 Container Diagram

Figure 5.2 illustrates the full deployment topology of the Multi-agent Preemptive Single-Shot architecture, including all containerised services, shared volumes and inter-service communication paths.

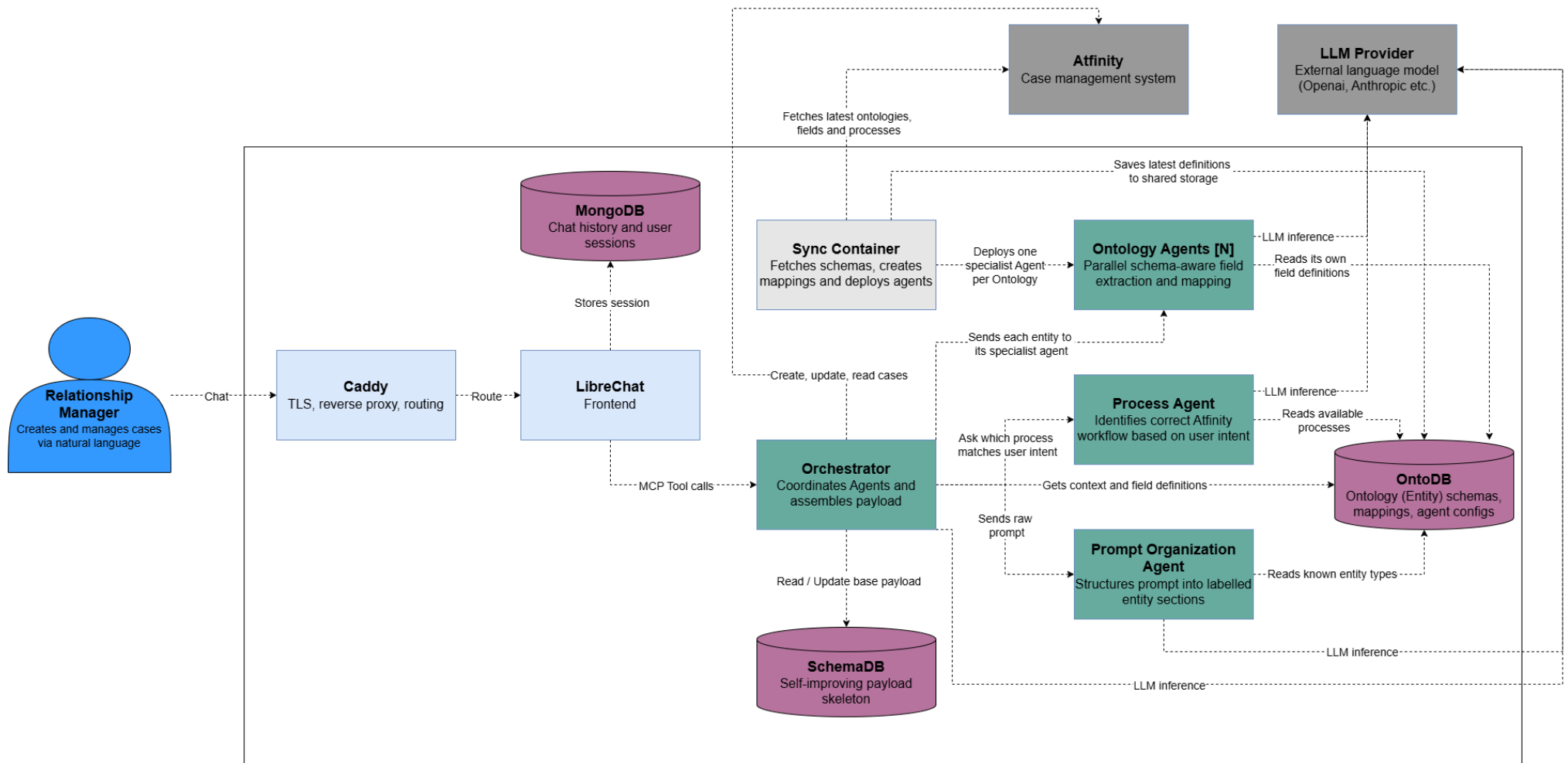


Figure 5.2: C4 Container Diagram: Multi-agent Preemptive Single-Shot Architecture

5.6.4 Components

The system comprises twelve distinct components: Caddy, LibreChat, MongoDB, the Orchestrator, the Prompt Organizer Agent, the Process Agent, the Ontology Agents, the Sync Container, the OntoDB volume, the SchemaDB volume, the Atfinity CMS and the LLM Provider. Each is described in turn below: first the components an incoming request passes through in sequence, then the supporting infrastructure, storage volumes and external services they all depend on. The rationale behind the selection of each key component, including the alternatives considered and the trade-offs accepted, is documented in the Architectural Decision Records in Section 5.7.

Reverse Proxy: Caddy

Caddy is an open-source web server and reverse proxy that sits at the outermost layer of the deployment. It handles all inbound HTTPS traffic from the Relationship Manager’s browser and is the first component any request reaches before entering the internal Docker network. Its primary responsibility in this system is TLS termination. Caddy accepts encrypted HTTPS connections from the public internet and forwards the decrypted requests over plain HTTP to the appropriate LibreChat container on the internal Docker network. This boundary means that no internal service needs to manage TLS certificates or handle encrypted transport directly.

A key operational advantage of Caddy over traditional reverse proxies is automatic certificate management. Caddy provisions and renews TLS certificates from Let’s Encrypt using the ACME protocol (Automated Certificate Management Environment) at startup, with no manual configuration required. The system is served under the `asoftware.ch` domain: the production environment is routed to `mas-preemptive.asoftware.ch` and the development environment to `mas-preemptive-dev.asoftware.ch`. Caddy distinguishes between these two environments via subdomain-based routing and forwards each request to the corresponding LibreChat container by container name on the shared Docker network.

Caddy is one of the two shared infrastructure services, alongside MongoDB, that is deployed once and shared across both the production and development environments on the OST VM. Without it, LibreChat would have to be exposed over an unencrypted HTTP port, which is not acceptable for a banking-context application that transmits client data.

Frontend: LibreChat

LibreChat is a self-hosted, open-source web application for large language models. It is the only surface the Relationship Manager interacts with directly. The RM enters natural-language prompts through a browser-based chat interface and LibreChat renders the system’s structured responses, including onboarding previews, confirmation summaries and case update results, as formatted conversation turns. In the present architecture, LibreChat fulfils three distinct roles simultaneously.

As the primary user interface, LibreChat provides the RM with a familiar conversational interface without exposing any system internals. The RM has no visibility into the Orchestrator, the agent pipeline or Atfinity’s REST API; all interactions are expressed in natural language and translated to structured operations by the system.

As an API gateway, LibreChat provides native support for the Anthropic API and forwards the RM’s messages to Claude, together with the complete manifest of MCP tools exposed by the Orchestrator. This conversational model is fixed to Claude via LibreChat’s native Anthropic endpoint and is independent of the per-component pipeline models described under LLM Provider below. Claude receives both the conversation history and the tool descriptions and reasons over them to decide which tool to invoke, with what arguments and in what order. No custom middleware is required to connect Claude’s reasoning to the system’s operational tools.

As the MCP client, LibreChat connects to the Orchestrator’s MCP server endpoint at startup, retrieves the tool manifest through the MCP capability handshake and forwards all of Claude’s tool invocations to the Orchestrator at runtime. The Orchestrator’s responses are returned to Claude, which incorporates them into its ongoing reasoning before producing a natural-language reply to the RM. The MCP protocol

is the mechanism by which the LLM’s reasoning capability is connected to the system’s operational tools without any tight coupling between the two.

LibreChat is deployed as a Docker container on the shared **onboarding-network**. Conversation histories, user credentials and session metadata are persisted in MongoDB. A `librechat.yaml` configuration file, mounted read-only into the container at startup, specifies the Orchestrator’s MCP server connection address and the Anthropic API credentials.

Persistent Storage: MongoDB

MongoDB serves as the data persistence layer for LibreChat. It stores conversation histories, user account credentials, session tokens and UI configuration metadata. MongoDB is not involved in any part of the multi-agent pipeline; it is exclusively a LibreChat dependency required to maintain state across browser sessions and page reloads. Without it, every browser refresh would clear the Relationship Manager’s conversation history, making the system unusable in any operational context.

MongoDB runs as a Docker container connected to the shared network and is one of the two shared infrastructure services alongside Caddy. A single MongoDB instance is shared between the production and development LibreChat deployments on the OST VM, providing a unified persistence layer that does not require duplication across environments.

Orchestrator: MCP Server

The Orchestrator is the central coordination component of the multi-agent system. Implemented as a FastMCP server in Python, it is the sole MCP endpoint exposed to LibreChat and the only component in the system that Claude calls directly. All coordination logic, all Atfinity API interactions and all agent dispatch decisions flow through it.

The Orchestrator registers and exposes ten MCP tools, each implemented in a dedicated module and registered with the FastMCP server at startup. The server exposes them as part of its MCP capability manifest, which LibreChat retrieves and forwards to Claude. The ten tools cover the full set of natural-language-accessible operations defined in Chapter 2:

- **preview_onboarding**: executes the full extraction pipeline and returns a structured preview to the RM without creating any case
- **create_client_onboarding**: runs the full MAS pipeline and submits the assembled payload to Atfinity
- **list_onboarding_processes**: queries the Process Agent for the best-matching Atfinity process for a given scenario
- **read_case**: fetches the live state of a case including all instances, roles and field values
- **update_case_field**: updates one or more fields in an existing case via natural language
- **add_case_instance**: creates a new party instance in a case, resolving the ontology type and any initial roles from the prompt
- **delete_case_instance**: removes a party instance; falls back to stripping all roles if Atfinity does not permit hard deletion
- **add_instance_role**: assigns an additional role to an existing instance, identifying the instance and role from natural language
- **remove_instance_role**: revokes a role from an existing instance
- **set_instance_roles**: replaces the full role set of an existing instance, computing the delta between current and desired roles and applying add and remove operations accordingly

For the `preview_onboarding` and `create_client_onboarding` tools, the Orchestrator coordinates the full multi-agent pipeline. It calls the Prompt Organizer Agent to restructure the RM's input, calls the Process Agent to determine the correct Atfinity onboarding process, fans out in parallel to all relevant Ontology Agents to extract field values, assembles the master payload in the predefined-object format Atfinity expects and submits it via a self-healing retry loop that contacts Ontology Agents for semantic corrections when Atfinity rejects specific fields. For read, update and instance management operations, the Orchestrator communicates with Atfinity's REST API directly, using targeted LLM calls only where natural-language resolution of instance identities or role keys is required.

To make routing decisions and determine which agents to invoke for a given case, the Orchestrator maintains an in-memory ontology directory called MemOrch, loaded from the `orch_context.yaml` file written by the Sync Container to the OntoDB volume. MemOrch holds a compact summary of each ontology's key, display name, description and supported roles. This compact context is what the Orchestrator passes to its routing LLM call rather than the full schema files, keeping the routing prompt small while still giving the model sufficient information to determine which ontologies are needed and how many instances of each a given case requires.

The Orchestrator reads from two shared Docker volumes. The OntoDB volume supplies the ontology context files and the agent manifest written by the Sync Container, which the Orchestrator uses for routing decisions and for resolving Ontology Agent service URLs at runtime. The SchemaDB volume stores the base schema skeleton and successful case payloads, which the Orchestrator uses to initialise the payload structure for each new submission. The Orchestrator exposes its MCP endpoint on port 8050 and is reachable to LibreChat over the shared Docker network. All LLM calls made directly by the Orchestrator are routed through LiteLLM and the active model is configured via the `LLM_MODEL` environment variable.

Prompt Organizer Agent

The Prompt Organizer Agent is a lightweight microservice responsible for restructuring the Relationship Manager's free-text prompt before it enters the extraction pipeline. In multi-party onboarding cases, a single prompt typically describes several individuals whose data is interleaved in an unstructured way. The RM might mention one person's name, then another's date of birth, then return to the first person's address, all within the same paragraph. Without preprocessing, the downstream Ontology Agents would need to reason across a mixed input containing data for multiple distinct individuals, which introduces systematic misattribution errors as described in Section 5.7.11.

The agent resolves this by reorganising the scattered prompt into clearly labelled, entity-separated sections. Each section begins with a structured header of the form `[Person - Full Name]` or `[Account]` and all information attributable to that entity is grouped under it. Shared contextual attributes that apply to all parties, such as investment horizon, risk profile and MiFID questionnaire answers, are explicitly duplicated across all relevant sections rather than placed only once, ensuring that no extraction agent receives an incomplete picture of its assigned entity. The agent never adds, infers or removes any information; it only reorganizes the original content. If the LLM call fails for any reason, the agent falls back to returning the original prompt unchanged, so the pipeline continues without interruption.

The agent exposes a single HTTP endpoint (`POST /organize`) and runs as a Docker container on the shared network. It loads the entity type catalogue from the `orch_context.yaml` file in the OntoDB volume so that its reorganization is aware of the actual ontology types defined in the connected bank's CMS. The active LLM model is configured via the `LLM_MODEL` environment variable, which allows a smaller, cost-efficient model to be assigned to this comparatively low-complexity preprocessing task.

Process Agent

The Process Agent, referred to in the codebase as the Process Determination Agent (PDA), is responsible for identifying which Atfinity onboarding process the Relationship Manager's request should trigger. This step is necessary because Atfinity organizes all case creation operations under configurable process definitions, each associated with a specific business partner type, a set of compliance rules and a set of party roles that the process creates automatically. Selecting the wrong process would result in the wrong compliance rules being applied and the wrong party structure being created, regardless of how accurately the field extraction performed downstream.

The agent maintains an in-memory process catalogue called MemPDA, loaded from a `processes_mapping.yaml` file written to the OntoDB volume by the Sync Container. Each entry in MemPDA contains the Atfinity process identifier, the process key, a human-readable display name, a natural-language description, a list of representative trigger phrases called intents, a key differentiator that distinguishes the process from structurally similar ones and the outcome ontology that Atfinity creates automatically when that process is started. The intents list is the primary signal for LLM-based matching: each entry is a purpose-written phrase that captures a typical way a Relationship Manager might phrase a request for that process type.

When the Orchestrator calls the agent's `POST /determine_process` endpoint, the agent constructs a structured prompt incorporating all MemPDA entries and the RM's reorganised input and instructs the LLM to identify the two or three most plausible candidate processes before selecting the single best match and assigning a confidence score between 0.0 and 1.0. If the returned confidence falls below the configured minimum threshold of 0.6, the agent returns no match and the Orchestrator terminates the pipeline, informing the RM that no process could be identified with sufficient certainty. On a successful match, the returned outcome ontology tells the Orchestrator which ontology Atfinity will create automatically as the case backbone, for example Account in a standard private banking onboarding. The Orchestrator excludes this ontology from the routing fan-out so that no duplicate instance is created in the submitted payload.

MemPDA auto-refreshes when the `processes_mapping.yaml` file on disk is modified by the Sync Container, so schema updates propagate to the agent without a container restart. The LLM model used by the Process Agent is configured via the `LLM.MODEL` environment variable.

Ontology Agents

Ontology Agents are the field extraction workers of the pipeline. Each agent is responsible for extracting structured field values for exactly one ontology from a focused sub-prompt delivered by the Orchestrator. In the Bank of Luxembourg configuration, 23 Ontology Agent containers are deployed concurrently, one per ontology, with each container named `ontology-agent-{key}-{env}` and listening on port 8080.

All Ontology Agents share the same codebase but are configured individually at deployment time. Each container receives its assigned ontology key via the `ONTOLOGY_KEY` environment variable and reads its per-ontology configuration file from the OntoDB volume at startup. This configuration file, written by the Sync Container, contains a system prompt tailored to the ontology's purpose. For the Person ontology, the system prompt includes explicit instructions for splitting full names into first and last name fields, mapping phone country codes from +XX notation to CMS enumeration keys and defaulting to natural person type when the prompt does not specify otherwise. For the Account ontology, the prompt enforces mandatory extraction of the business partner type and account type fields, instructing the agent to read the description of every available enumeration option before selecting one rather than defaulting to the most generic match.

Each Ontology Agent exposes two HTTP endpoints. The `POST /process` endpoint receives a focused sub-prompt, reads the full field schema for its assigned ontology from the OntoDB volume and invokes its LLM with strict extraction instructions to return a flat JSON dictionary of field key-value pairs. Enumeration fields must be returned as the internal enum key rather than the human-readable label; monetary amounts must conform to the structured format `{decimal_number, unit, item}`; dates must be converted to ISO 8601 regardless of the input notation. The `POST /correct` endpoint is called by the Orchestrator during the self-healing retry loop: it receives the original sub-prompt, the specific fields that Atfinity rejected and their invalid values and uses the rejection context together with the schema's enumeration constraints to return corrected values.

The Orchestrator invokes all relevant Ontology Agents in parallel using a thread pool executor, so the total extraction time for a multi-party case scales with the slowest individual agent response rather than with the cumulative sum of all agent latencies. Each Ontology Agent's LLM model is set at deployment time by the Sync Container via the `ONTOLOGY_LLM.MODEL` deployment variable, which is forwarded as `LLM.MODEL` into each spawned container, so a single configuration change propagates uniformly to all agents.

Sync Container

The Sync Container is a background service that performs two distinct responsibilities, both critical to keeping the system's schema context aligned with Atfinity's live CMS configuration.

Its first responsibility is schema synchronisation. On startup, the container authenticates with Atfinity's REST API and retrieves the full set of available processes, ontologies and field definitions for the connected tenant. The ontology mapping and process mapping scripts run in parallel, as they are fully independent. The container transforms the retrieved data into structured YAML files and writes them to the OntoDB shared volume. These files serve as the primary knowledge source for the Process Agent's MemPDA, the Prompt Organizer Agent's entity type catalogue, the Orchestrator's MemOrch routing context and each Ontology Agent's field extraction schema. Because the Sync Container runs once and exits, schema re-synchronisation requires an explicit re-trigger, either through a tagged CI/CD release or a manual re-run of the container, rather than on an automatic schedule; the rationale behind this design choice is discussed in Section 6.2.

Its second responsibility is Ontology Agent deployment. On every startup, the Sync Container builds a fresh Docker image from the shared Ontology Agent source code. The image is always rebuilt rather than reused, because Docker's layer cache makes a no-change build complete in seconds and rebuilding guarantees that any code changes delivered via the CI/CD pipeline are immediately reflected in the spawned containers. After building, the Sync Container stops and removes any existing Ontology Agent containers and launches one new container per ontology discovered in the CMS. This deployment is implemented via the Docker socket, which is mounted into the Sync Container at runtime. The Sync Container discovers the name of the OntoDB volume and the Docker network it belongs to by introspecting its own container metadata at runtime, so no hardcoded volume or network names are required in the deployment configuration.

All dynamically spawned Ontology Agent containers are attached to the shared **onboarding-network** and registered in an **agents_manifest.yaml** file written to the OntoDB volume. The manifest records each agent's container name and HTTP service URL in the form `http://ontology-agent-{key}-{env}:8080`, which the Orchestrator reads at runtime to resolve agent endpoints without any hardcoded addresses.

OntoDB Volume

The OntoDB is a Docker named volume that serves as the shared schema knowledge base for all components that need access to Atfinity's data model. It is written exclusively by the Sync Container and mounted read-only by all consuming services: the Orchestrator, the Process Agent, the Prompt Organizer Agent and all deployed Ontology Agent containers.

The volume holds the following artefacts, all produced by the Sync Container: one **ontology_{key}.yaml** file per ontology containing the full field schema with field keys, display names, data types and enumeration constraints; a **processes_mapping.yaml** file with all process definitions, their intent phrases and their outcome ontologies; an **orch_context.yaml** file with a compact ontology summary used by the Orchestrator's MemOrch and the Prompt Organizer Agent's entity type catalogue; an **agents_manifest.yaml** registering the HTTP service URLs of all deployed Ontology Agent containers; and one **agent_{key}.yaml** per ontology containing the per-agent system prompt and configuration used at extraction time.

Because all consumers mount the same volume, a schema update written by the Sync Container is immediately visible to all agents on their next read. The mtime-watching auto-refresh mechanism used by MemPDA, MemOrch and the Prompt Organizer Agent exploits this property to pick up process and ontology changes within seconds of the Sync Container writing them, without requiring any container restarts or explicit inter-service synchronisation.

SchemaDB Volume

The SchemaDB is a Docker named volume used exclusively by the Orchestrator. It stores two categories of data. The first is **base_schema.json**, a structural skeleton of the Atfinity case creation payload that the Orchestrator uses as the starting frame for every payload assembly. The second is a **successful_payloads/** directory containing a timestamped archive of every payload that Atfinity has

accepted with an HTTP 200 response.

The learning mechanism works as follows. On first startup the Orchestrator writes a minimal default skeleton to the volume, containing the required top-level payload structure but no predefined-object entries. When a case creation succeeds, the Orchestrator archives the accepted payload under a timestamped filename and updates the base schema by extracting the structural shape of the successful payload, keeping field keys but clearing their values and writing the result as the new baseline. On the next submission, the Orchestrator initialises the payload from this updated skeleton, carrying over structural defaults such as the `preferred_language_key` field that remain consistent across submissions. This mechanism eliminates a class of structural rejections that would otherwise occur if the skeleton were derived from documentation alone, since certain top-level fields required by the specific tenant configuration are not discoverable through Atfinity’s schema API.

The archived payloads directory provides a complete timestamped audit trail of all accepted submissions, independently useful for debugging validation errors and for post-hoc analysis of extraction quality across evaluation runs.

Backend Integration: Atfinity CMS

Atfinity’s CMS is the authoritative backend for all client onboarding data. It is an externally operated system, deployed and managed by Atfinity AG and accessed by the Orchestrator exclusively over the network through Atfinity’s REST API. The Orchestrator interacts with four categories of endpoints: case creation via a bulk POST request using a fully assembled `predefined_objects` payload; case retrieval via GET requests for field data and instance lists; field update via PATCH requests for individual field modifications; and instance and role management via POST and DELETE requests for party instance creation and removal and PUT requests for role assignments. All authentication credentials, including the API key and the base URL, are injected into the Orchestrator at runtime via environment variables and are never stored in the codebase.

In the present work, the system is connected to a dedicated sandbox tenant provisioned as a representative copy of the Bank of Luxembourg’s CMS configuration. This tenant exposes 23 ontologies, 30 processes and approximately 25 000 fields across the entire schema and serves as the integration target for all evaluation runs described in Chapter 7.

LLM Provider

The LLM Provider is the external AI inference service to which all language model calls within the system are routed. It is not part of the system boundary and is operated independently of the deployment infrastructure; it is accessed exclusively over HTTPS from within the Docker network by the components that require it.

Four components issue LLM calls: the Orchestrator for ontology routing and targeted instance resolution, the Process Agent for process intent classification, the Prompt Organizer Agent for prompt restructuring and each Ontology Agent for field extraction and semantic correction. Every one of these calls is routed through LiteLLM, a provider-agnostic completion library that translates a standardized internal API call into the provider-specific wire format at runtime. The supported providers in the present deployment are Anthropic, with the Claude model family and OpenAI, with the GPT model family. The active model for each component is specified independently via a dedicated `LLM_MODEL` environment variable using LiteLLM’s provider-prefixed model string format, for example `anthropic/claude-sonnet-4-5` or `openai/gpt-4o-mini`. The Orchestrator, the Process Agent, the Prompt Organizer Agent and the Ontology Agents can therefore each be assigned a different model without any code modifications, which is a direct prerequisite for the efficiency evaluation in Chapter 7.

5.6.5 Context Scoping per Agent

The extraction quality of the pipeline depends directly on each agent receiving a precisely bounded context: the minimum information required to perform its task correctly, with no contaminating data from other entities or ontologies. Context scoping in the Multi-agent Preemptive Single-Shot architecture operates across two orthogonal dimensions: schema scoping, which controls the set of fields presented

to the extraction LLM and prompt scoping, which controls the fragment of the Relationship Manager’s input each agent reasons over.

Schema Scoping. Each Ontology Agent is statically bound to a single ontology and reads its schema from its assigned `ontology_{key}.yaml` file in the OntoDB volume at extraction time. The Orchestrator never transmits schema data to an agent; each agent independently resolves its own schema. For the Bank of Luxembourg configuration, this means the Person Agent reads only the Person schema, the Account Agent reads only the Account schema and so on, independently of the 22 other ontologies present in the tenant. A single ontology schema typically spans 50 to 300 fields. This constrains the extraction LLM’s reasoning surface to the fields it is authorised to populate, eliminating cross-ontology field key confusion and reducing hallucination risk. For ontologies whose schema exceeds the model’s effective context window, the schema is partitioned into fixed-size windows of 30 fields each, with each window dispatched as a separate parallel LLM call (Section 6.5.1). This extends exact schema scoping to schemas of arbitrary size without imposing a single enormous context on one LLM invocation.

Prompt Scoping. While schema scoping restricts the field universe available to each agent, prompt scoping restricts the client data each agent is permitted to reason over. In a multi-party case, the original Relationship Manager input describes several individuals in a single unstructured paragraph. Without preprocessing, an Ontology Agent tasked with extracting fields for one individual would see the personal data of all other individuals simultaneously, creating systematic misattribution risk. The two-stage prompt scoping mechanism described in Section 5.7.11 resolves this before any Ontology Agent is invoked. In the first stage, the Prompt Organizer Agent restructures the raw input into entity-labelled sections, one per individual, with shared contextual attributes such as investment horizon and MiFID questionnaire answers explicitly duplicated into every relevant section. In the second stage, the Orchestrator’s routing step decomposes the reorganised input into per-entity sub-prompts. Each sub-prompt contains only the data attributable to one specific individual, identified by full name, together with all shared contextual attributes. The resulting sub-prompt is the sole natural-language input the corresponding Ontology Agent receives; it has no access to the data of any other individual described in the original prompt.

The combination of schema scoping and prompt scoping reduces every Ontology Agent invocation to an isolated extraction problem: one ontology schema against one entity’s data. This is the structural basis for the zero cross-entity misattribution rate observed in the TEST3 evaluation results under GPT-5.4, where all six simultaneous role-holders were correctly extracted with 100% precision across every archetype (Section 7.7.1).

5.6.6 Case Creation Workflow

The Multi-agent Preemptive Single-Shot pipeline defines the ordered sequence of interactions between all system components, from the Relationship Manager’s natural-language prompt to a confirmed case in Atfinity. The pipeline operates across three distinct phases, each documented below with its sequence diagram followed by a detailed description. The background phase runs independently of any user interaction. It keeps all agent knowledge sources aligned with Atfinity’s live CMS schema and deploys the Ontology Agent containers that the extraction step dispatches to in parallel. The preview phase runs the prompt reorganisation, process determination, outcome extraction and ontology routing steps, returns a structured review of these results to the Relationship Manager and halts before the parallel field extraction of the remaining entities and the creation of any case. The create phase resumes from the exact state the preview left off, reuses cached results to eliminate redundant LLM calls and submits the assembled payload to Atfinity atomically. Because the complete flow spans five sequence diagrams, each figure covers a distinct segment of the pipeline; the description following each figure explains that segment’s mechanics and how it connects to the segments before and after it.

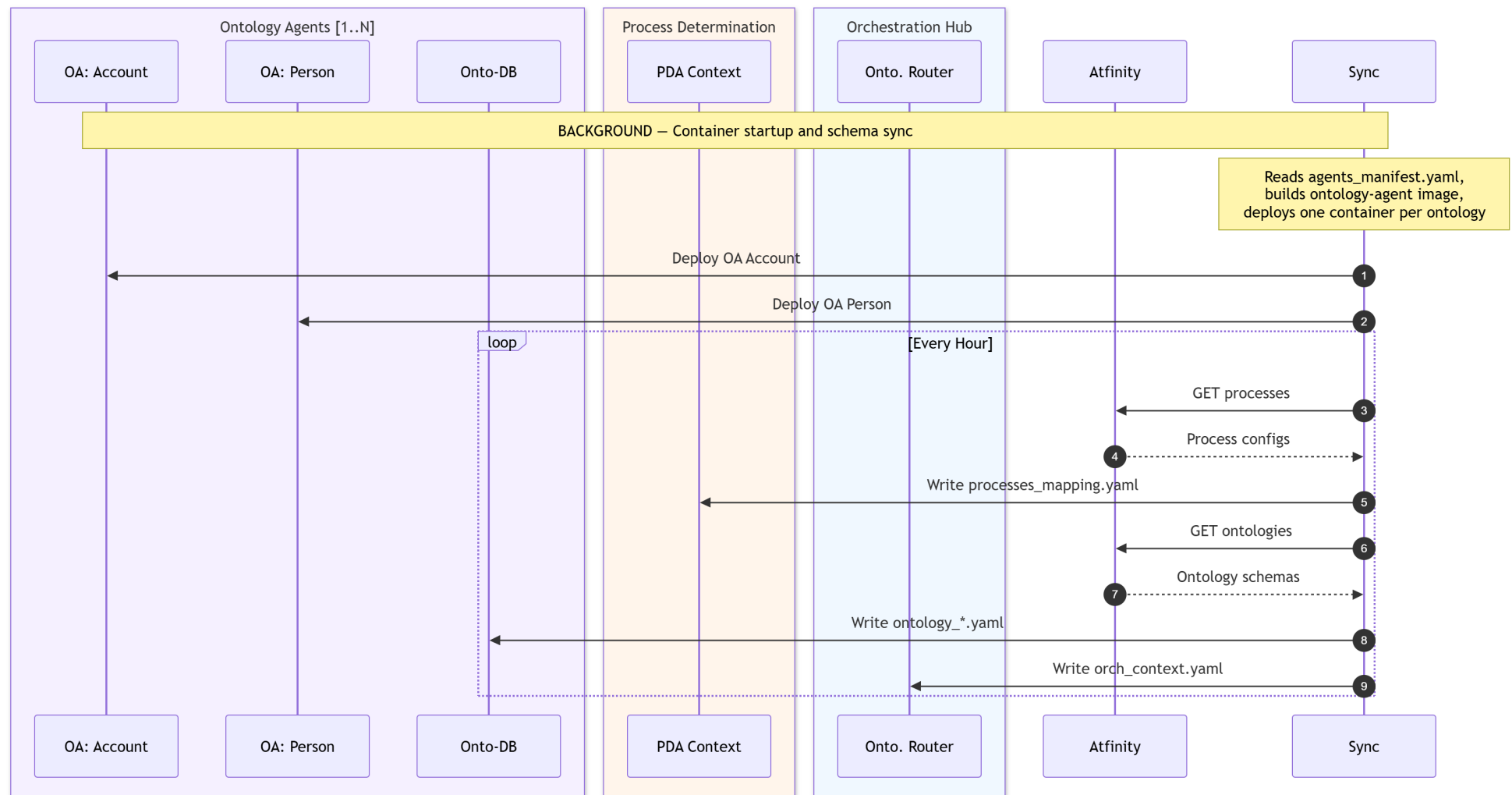


Figure 5.3: Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Background phase: Sync Container startup and schema synchronisation

Background Phase: Sync Container Startup and Schema Synchronisation

Figure 5.3 illustrates the background phase. This phase runs once on Sync Container startup and is the prerequisite for every other phase. Without it completing successfully, the Ontology Agent containers that the create phase fans out to would not exist and neither the Process Agent nor any Ontology Agent would hold a current copy of Atfinity’s CMS schema.

On startup, the Sync Container reads the existing agents manifest, always rebuilds a fresh Ontology Agent Docker image from the shared agent source code to ensure that code changes delivered via the CI/CD pipeline are immediately picked up and launches one container per ontology defined in Atfinity’s CMS. Container names follow the pattern `ontology-agent-{key}-{env}` and all spawned containers are attached to the shared `onboarding-network` with read-only access to the OntoDB volume.

The Sync Container authenticates with Atfinity’s REST API and retrieves the full set of process definitions and ontology schemas. It writes the process definitions to `processes_mapping.yaml`, which the Process Agent’s MemPDA reads on its next access via the mtime-watch mechanism. It writes each ontology schema to an individual `ontology-{key}.yaml` file in the OntoDB volume, which each Ontology Agent reads at extraction time. A compact `orch_context.yaml` summary is also written for the Orchestrator’s MemOrch and the Prompt Organizer Agent’s entity type catalogue. Because all consumers watch the file modification time of these outputs, schema updates propagate to all agents without container restarts. Re-synchronisation against Atfinity requires an explicit re-trigger via CI/CD, as described in Section 5.6.4.

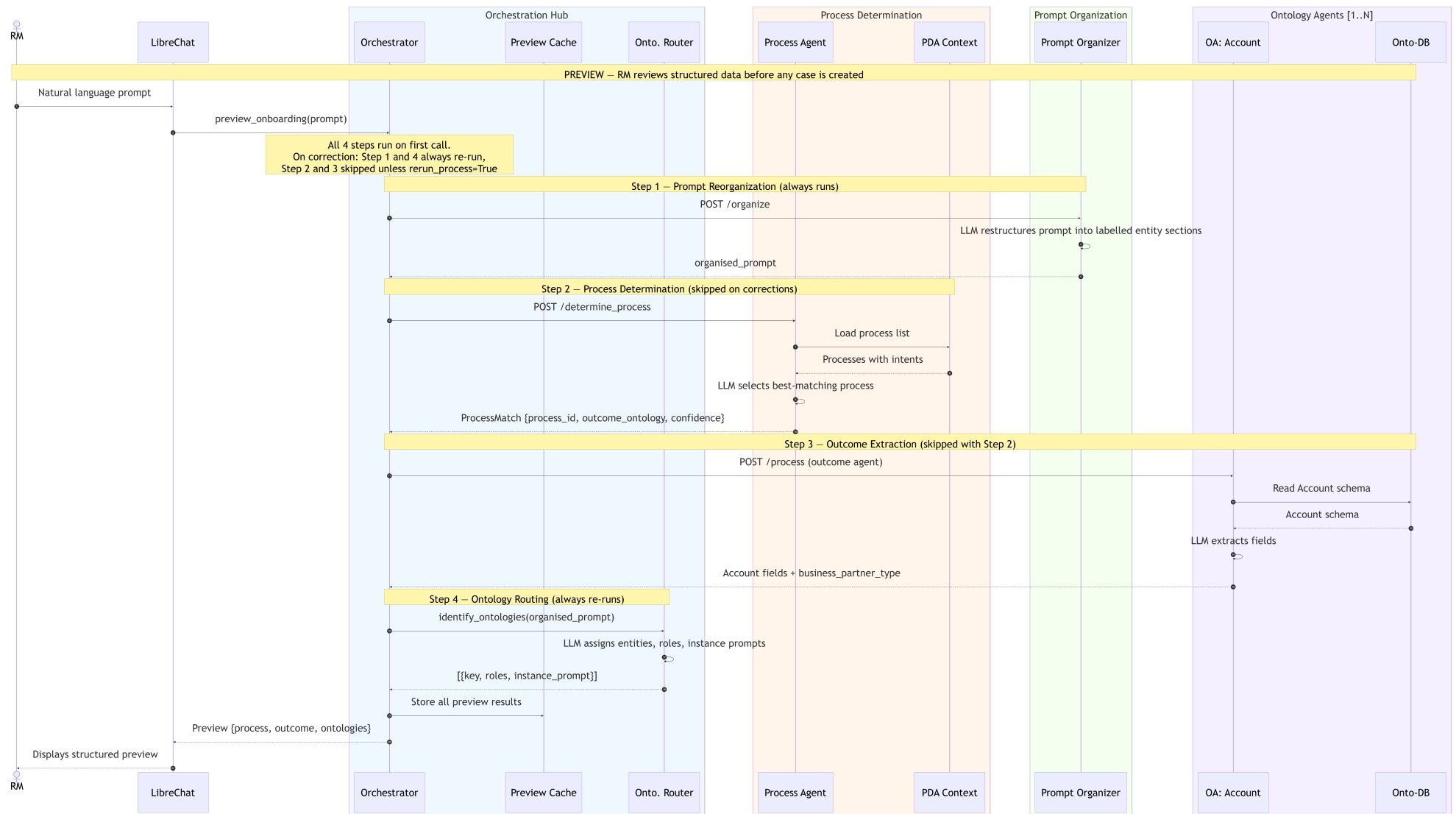


Figure 5.4: Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Preview phase

Preview Phase

Figure 5.4 shows the preview phase, which is the first user-triggered segment of the pipeline. Building on the agent infrastructure and schema knowledge established in the background phase (Figure 5.3), the preview phase is initiated when the Relationship Manager calls the `preview_onboarding` MCP tool via LibreChat. Its purpose is to expose the pipeline’s extraction results for review before any case is committed to Atfinity. The RM can review the reorganised client data, the matched process, the identified business partner type and the planned entity list and then confirm, correct or abort, catching misextractions while they are still cheap to fix rather than after submission. All executed step results are stored in the preview cache for reuse in the create phase, so approving a preview costs zero additional LLM calls when the Relationship Manager confirms immediately.

On a first call, all four preview steps run in full. On correction calls, when the Relationship Manager has reviewed a preview and provided updated data, the Orchestrator minimizes redundant computation. The Prompt Organizer and the Ontology Router always re-run, since the organised text and the entity structure may change with any data update. The Process Agent is skipped unless the RM explicitly indicates that the process or workflow is wrong, in which case the `rerun_process` flag is passed. The Outcome Agent follows the Process Agent exactly. It re-runs only when the process is re-run, since the outcome ontology is determined by the process.

Preview Step 1: Prompt Reorganisation (always runs). The Orchestrator sends the raw prompt to the Prompt Organizer Agent (`POST /organize`). The agent uses an LLM to rewrite the scattered input into clearly labelled sections, one per person or legal entity, so that all information about the same individual appears together. Attributes that apply to all parties, such as investment horizon and MiFID questionnaire answers, are intentionally duplicated across every relevant section so no extraction agent receives an incomplete picture. If the LLM call fails for any reason, the agent returns the original prompt unchanged and the pipeline continues without interruption.

Preview Step 2: Process Determination (skipped on corrections unless `rerun_process=True`). The Orchestrator sends the raw prompt to the Process Agent (`POST /determine_process`). The agent compares the prompt against all known Atfinity processes, each described by its purpose, typical trigger phrases and a key differentiator that distinguishes it from similar processes. It returns the best match together with a confidence score between 0.0 and 1.0. If the score falls below 0.6, the preview stops immediately and the RM is informed that the process could not be identified with sufficient certainty. On a successful match, the response also includes the outcome ontology, which is the primary entity type Atfinity creates automatically when that process is started, for example Account in a standard private banking onboarding.

Preview Step 3: Outcome Extraction (follows Step 2). The Orchestrator calls the Ontology Agent responsible for the outcome ontology identified in Step 2 (`POST /process`). That agent reads its schema from OntoDB and extracts the relevant fields from the organised prompt, including the `business_partner` field that specifies the type of client entity. The Orchestrator translates this internal enum key into a human-readable label and includes it in the preview so the RM can confirm the correct client type was identified. The result is saved in the preview cache and reused in the create phase, so this agent is never called a second time for the same request.

Preview Step 4: Ontology Routing (always re-runs). The Orchestrator runs an internal routing step using an LLM and the compact MemOrch context, which holds a brief summary of every ontology in the system. The LLM determines which ontology types are needed, how many instances of each are required and which roles each instance should hold. The outcome ontology from Step 3 is excluded here since it has already been handled. For each identified instance, the router produces a focused sub-prompt containing only that individual’s data. These scoped sub-prompts are the key mechanism for preventing data from one person being assigned to another in multi-party cases, as described in Section 5.7.11. This step always re-runs on corrections because any change to the RM’s input can affect the entity count, role assignments or per-instance data boundaries.

This routing step is conceptually distinct from Step 1’s prompt reorganisation, even though both appear to “split the prompt by entity”. Step 1 is a purely textual regrouping. It has no knowledge of Atfinity’s ontologies or roles and only answers which sentences belong to the same individual. This

routing step instead answers what each grouped individual actually is in Atfinity's data model, which ontology type, how many instances and which role, a mapping that depends on the MemOrch context and is therefore only possible at the Orchestrator level. The two steps are sequential and complementary. Step 1 produces the clean, entity-grouped input this step needs to route reliably, while this step turns that grouping into the concrete ontology and role assignments the extraction stage depends on.

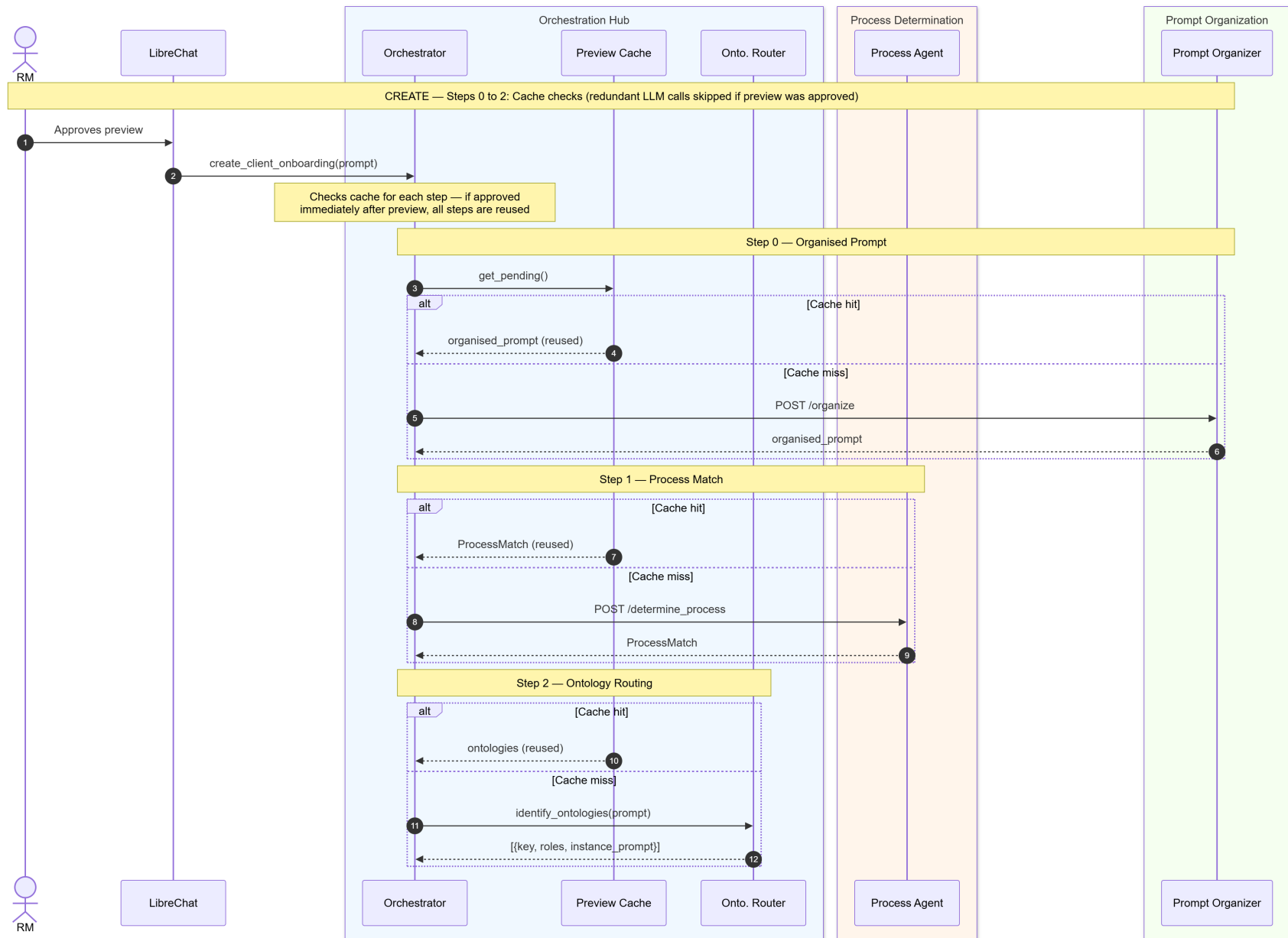


Figure 5.5: Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Create phase, Steps 0 to 2: cache verification

Create Phase, Steps 0 to 2: Cache Verification

Figure 5.5 shows the opening of the create phase. Continuing directly from the preview phase (Figure 5.4), the create phase is initiated when the Relationship Manager approves the preview and calls `create_client_onboarding` via LibreChat. Steps 0 through 2 are cache-verification steps. For each step the Orchestrator first attempts to retrieve the result from the preview cache. If the RM approved the preview immediately before this call, all three steps will be cache hits and no additional LLM calls are made, directly reducing both pipeline latency and token consumption. Cache lookup uses two mechanisms: a pending-slot that always holds the most recent preview and requires no prompt-string matching and a SHA-256 hash-keyed fallback that covers cases where the pending slot was cleared by a container restart between the preview and the confirmation call. A direct call to `create_client_onboarding` without a prior preview will find all cache lookups as misses and execute every step in full.

Step 0: Organised Prompt. The Orchestrator checks the pending slot of the preview cache for an organised prompt. On a cache hit, the previously reorganised text is reused directly. On a cache miss, the Orchestrator calls the Prompt Organizer Agent’s `POST /organize` endpoint with the raw prompt.

Step 1: Process Match. The Orchestrator checks the cache for a process match result. On a cache hit, the cached `ProcessMatch` is reused, including the process identifier, display name and outcome ontology key. On a cache miss, the Orchestrator calls the Process Agent’s `POST /determine_process` endpoint. A failure to return a confident match terminates the pipeline immediately with an error message to the Relationship Manager.

Step 2: Ontology Routing. The Orchestrator checks the cache for the identified ontologies list. On a cache hit, the cached list of `{key, roles, instance_prompt}` entries is reused, preserving the exact per-instance sub-prompts that the Relationship Manager reviewed in the preview (Section 5.7.12). On a cache miss, the Orchestrator calls its routing LLM to re-identify the required ontologies and per-instance sub-prompts from the organised prompt, excluding the outcome ontology.

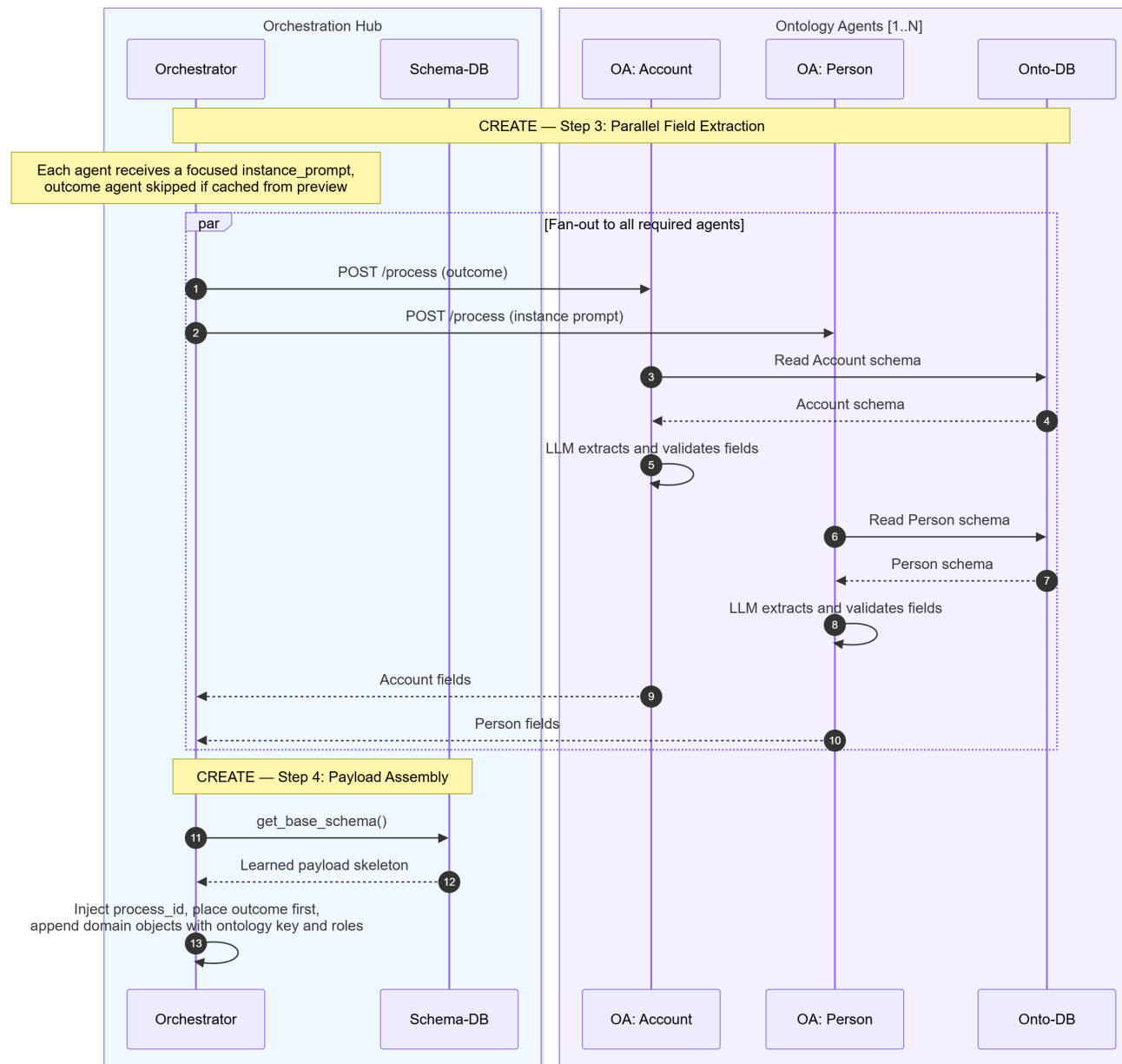


Figure 5.6: Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Create phase, Steps 3 and 4: parallel extraction and payload assembly

Create Phase, Steps 3 and 4: Extraction and Assembly

Figure 5.6 shows how the routing information confirmed in Steps 0 to 2 is converted into a complete Atfinity payload. With all cache lookups resolved, the pipeline transitions from coordination into active extraction and assembly.

Step 3: Parallel Field Extraction. The Orchestrator fans out in parallel to all required Ontology Agent containers using a `ThreadPoolExecutor` (Section 5.7.8), implementing the Parallel Fan-Out/Fan-In pattern [13]. Each task submits a `POST /process` request to the corresponding agent with the focused `instance_prompt` for that entity. The agent independently reads its full ontology schema from the OntoDB volume, invokes its LLM with the per-ontology extraction instructions from its agent configuration file and returns a flat JSON dictionary of field key-value pairs. The outcome ontology agent is included in the fan-out with the full organised prompt as its input; if its extraction was cached from the preview, the agent call is skipped and the cached result is prepended to the collection. A 300-second HTTP timeout is applied per agent to accommodate large ontology schemas such as Person, which require multiple sequential LLM calls per schema chunk and can individually take up to 60 seconds. A failure in any individual agent returns an empty field set for that ontology without cancelling or delaying the remaining agents.

Step 4: Payload Assembly. With all extraction results collected, the Orchestrator assembles the master `predefined_objects` payload. It loads the base schema skeleton from the SchemaDB volume, which carries over structural defaults learned from prior successful submissions (Section 5.7.10) and injects the process identifier into the `process_id` field. The outcome ontology fields are placed first in the `predefined_objects` array without an `ontology` key, as Atfinity expects the outcome instance at the top of the array and creates it automatically. All remaining domain objects follow in sequence with their explicit `ontology` key and the role assignments determined in Step 2.

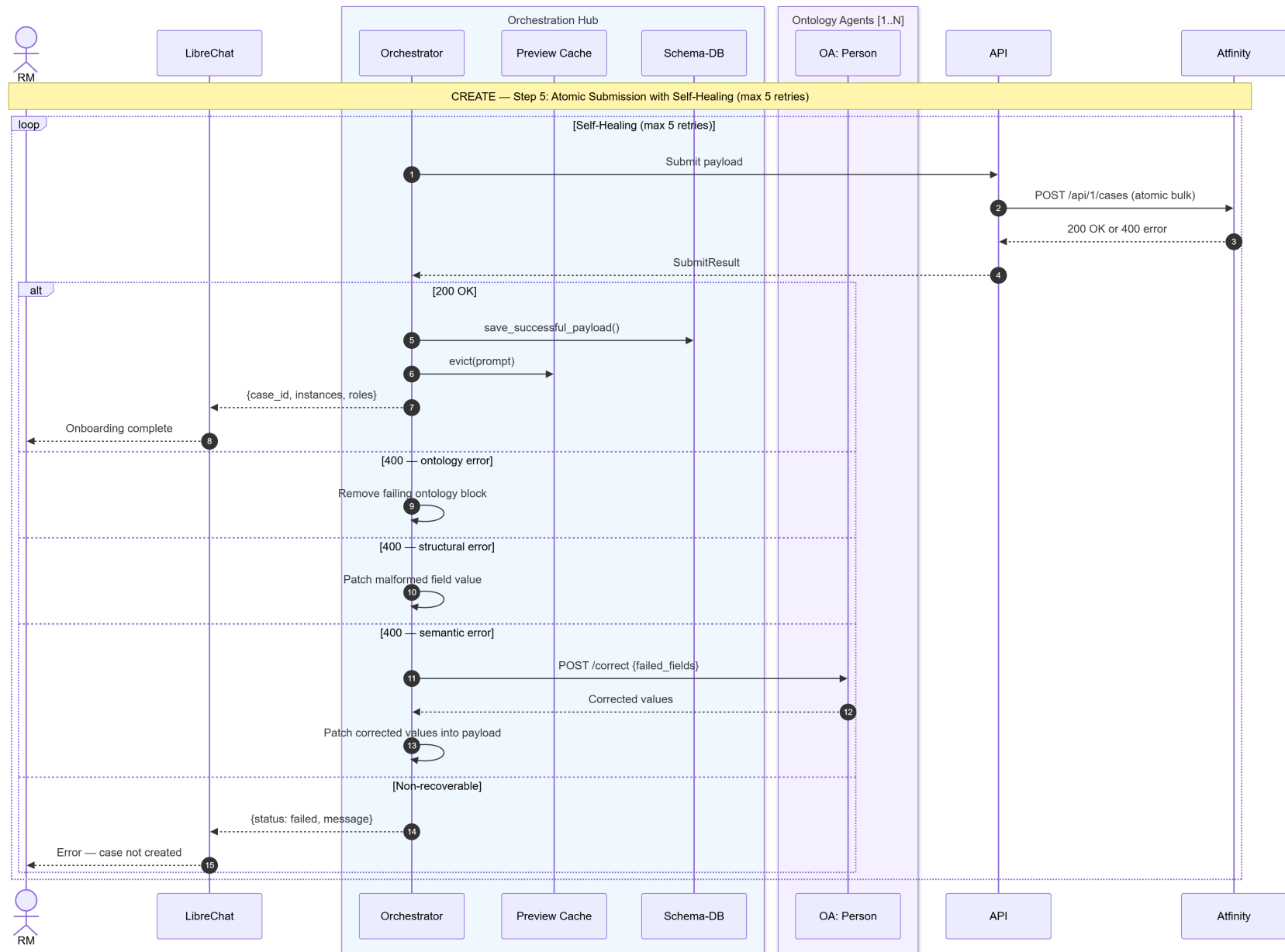


Figure 5.7: Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Create phase, Step 5: atomic submission with self-healing

Create Phase, Step 5: Atomic Submission with Self-Healing

Figure 5.7 shows the final step of the pipeline. The assembled payload from Step 4 is submitted to Atfinity’s `POST /api/1/cases` endpoint in a single atomic request (Section 5.7.2). The submission is atomic in Atfinity’s model. The entire payload is either accepted with HTTP 200 and all party instances are created simultaneously or it is rejected entirely with HTTP 400 containing per-field error details. No partial case is created on rejection. The Orchestrator runs a self-healing retry loop for up to five attempts before declaring failure (Section 5.7.9). This implements an externally-triggered variant of the Reflection/Self-Evaluation Loop pattern [13]: instead of the agent autonomously reviewing its own output, Atfinity’s validation layer acts as the evaluator and each rejection drives the next correction cycle.

On each rejection, the Orchestrator classifies every failed field into one of three categories. Ontology-level errors, such as an unexpected duplicate outcome instance triggered by a misconfigured `predefined_objects` array, cause the entire offending ontology block to be removed before the next retry. Field-level structural errors, where a value is technically malformed regardless of its content, for example a list submitted as a string with bracket notation, are patched directly by the Orchestrator using a deterministic reformatting rule or stripped from the payload if reformatting cannot resolve the issue. Field-level semantic errors, where a value is syntactically valid but incorrect in content, for example an unrecognised enumeration key, are routed back to the responsible Ontology Agent’s `POST /correct` endpoint along with the original prompt and the specific rejection context; the agent re-runs its LLM using the rejection information and the schema’s enumeration constraints to return corrected values. Correction requests to multiple agents are dispatched in parallel using a second `ThreadPoolExecutor`. If an agent’s correction call fails or returns null for a field, that field is stripped from the payload to allow the submission to proceed. If Atfinity returns an error containing no identifiable field or ontology reference, the loop breaks immediately since resending an identical payload cannot succeed.

On a successful submission, Atfinity returns the created case identifier and the full list of instances it created. The Orchestrator archives the accepted payload to the SchemaDB via `save_successful_payload()` to update the structural skeleton for future submissions, evicts the preview cache entry for this prompt and classifies the returned instances into two groups: user-created instances whose names were derived from the Relationship Manager’s prompt and auto-created instances that Atfinity generated to satisfy the compliance-driven role requirements of the matched process. Auto-created instances are identified by the naming pattern `<OntologyName> <id>`, for example “Person 2549”, which is the convention Atfinity uses for placeholder instances it creates without user-provided data. The two-group classification is returned to LibreChat so the Relationship Manager is informed both of the persons successfully onboarded and of any compliance-required placeholder records that Atfinity created and that may need to be populated with additional data.

5.6.7 Quality Attributes

The following paragraphs show how the architectural decisions described in this chapter address the non-functional requirements defined in Chapter 2 and the broader structural quality properties of the system.

NFR-1: Performance (≤ 120 s)

The acceptance threshold is addressed by two complementary mechanisms. The hybrid sequential-parallel execution pattern (Section 5.7.8) concentrates the parallelism where it has the greatest latency impact. The extraction phase fans out to all required Ontology Agents simultaneously, bounding total extraction time to that of the slowest individual agent rather than the cumulative sum. For a three-party case requiring three Person agent invocations of approximately 45 seconds each, this reduces the extraction contribution from approximately 135 seconds to approximately 45 seconds. The preview cache (Section 5.7.12) eliminates all redundant LLM calls on the approval path. When the Relationship Manager confirms an approved preview immediately, the cache-verification steps of the create phase, Steps 0 through 2 (Section 5.6.6: organised prompt, process match and ontology routing), are skipped entirely and the confirmation step reduces to the parallel field extraction of the identified entities, payload assembly and a single Atfinity API call.

NFR-2, NFR-3, NFR-4: Extraction Quality (Precision ≥ 0.90 , Recall ≥ 0.90 , F1-Score ≥ 0.85)

Three architectural mechanisms work together to meet these extraction quality targets. The two-stage prompt scoping strategy (Section 5.7.11) first reorganises multi-party inputs into entity-labelled sections and then produces a focused per-entity sub-prompt for each Ontology Agent, eliminating cross-entity data contamination before any extraction begins. Per-ontology agent scoping (Section 5.7.5) further limits each extraction agent's context to the schema of a single ontology, preventing the cross-ontology field confusion that a single large-context prompt would risk. Where Atfinity's validation layer rejects a submitted value as semantically incorrect, the self-healing retry loop (Section 5.7.9) routes that specific field back to the responsible Ontology Agent for LLM-assisted correction, preventing isolated extraction errors from causing complete case creation failures.

NFR-5: Efficiency (LLM calls, token usage and cost per case)

NFR-5 does not define an acceptance threshold; it defines the metrics that must be logged and compared across model configurations. The pipeline instruments every LLM call with its model name, prompt token count, completion token count and USD cost, aggregated per agent and per test case. Two architectural decisions directly influence what these measurements show. The LiteLLM abstraction (Section 5.7.3) allows each agent role to be assigned a different model via a single environment variable, making it possible to compare the cost and accuracy trade-offs across the four evaluated model configurations without code changes. The preview cache (Section 5.7.12) further reduces token consumption on the approval path. When the Relationship Manager confirms a preview immediately, no additional LLM calls are issued for the create phase, reducing the per-case token cost relative to a pipeline without caching.

NFR-6: Usability

The requirement that the Relationship Manager interacts exclusively in natural language, without supplying Atfinity's internal field identifiers, ontology keys, instance IDs or process numbers, is addressed at two levels. LibreChat (Section 5.7.13) provides the conversational interface through which the RM communicates; all ten MCP tools accept free-text natural-language prompts and return plain-language summaries, with no technical fields exposed to the user. The two-stage prompt scoping strategy (Section 5.7.11) additionally accepts any unstructured input format and reorganises it before processing, so the RM is not required to follow any specific prompt template or structure.

Scalability

Although scalability is not stated as a discrete NFR, it is a structural requirement imposed by the variable-ontology-count constraint of the problem. Dynamic agent deployment (Section 5.7.7) ensures that the number of active Ontology Agent containers scales automatically with the number of ontologies in the

connected CMS, so adding or removing an ontology in Atfinity requires no changes to any deployment manifest or source code. Per-ontology scoping (Section 5.7.5) ensures extraction quality does not degrade as the total schema size grows, since each agent’s reasoning context remains bounded to a single ontology regardless of the overall CMS scale.

Data Privacy and Security

Data privacy and security were not within the scope of this thesis and were not formally evaluated. The following observations are included solely to document the implicit constraints of the banking context and the deployment decisions they influenced. The OST VM deployment (Section 5.7.1) keeps all processing and evaluation data within a controlled institutional environment with no dependency on developer workstations. Caddy handles HTTPS termination at the network boundary (Section 5.7.14), ensuring that client data transmitted through the browser interface is encrypted in transit. All environment-specific secrets are injected at deployment time via GitLab Secure Files and are never committed to the repository. The LiteLLM abstraction (Section 5.7.3) further supports data-residency compliance. The active model for any agent role can be switched to a locally deployable variant without modifying pipeline logic, allowing all client data to be processed entirely on-premises when required by regulatory constraints.

5.7 Architectural Decision Records

Architectural decisions throughout this project were formed through iterative development and evaluation against predefined onboarding scenarios. Rather than committing to a fixed design upfront, each architecture version was tested against expected behaviour and the findings directly shaped the next iteration. The ADRs below document the key decisions made during the bachelor thesis phase, capturing the context, rationale and trade-offs considered at each decision point.

5.7.1 ADR01: OST VM as Exclusive Deployment Target with Automated CI/CD Pipeline

ADR ADR01
Author Arnel Veladzic
State Accepted
Related Deployment Environment

Decision Record Running the evaluation described in Chapter 7 on different developer machines would make the results incomparable. Differences in CPU performance, available memory and network distance to Atfinity’s sandbox tenant alone would shift the measured latency and cost numbers, regardless of how the system itself behaves. The decision was therefore made to run all integration testing, pilot operation and evaluation exclusively on one fixed machine, the OST VM, so that every run executes under the same hardware and network conditions and the results can be compared directly.

Concentrating everything on a single shared VM only works if deployments to it are reliable and repeatable, so the decision was also made to automate them through a GitLab CI/CD pipeline rather than deploying by hand. The pipeline has four stages, infrastructure, build, evaluate and deploy, mapped to two branch-specific environments on the same VM. The `dev` branch deploys to the development environment; the `main` branch deploys to the production environment. The build stage compiles Docker images for the Orchestrator, Process Agent and Prompt Organizer Agent on branch pushes; the Sync Container image is built separately on tag events to isolate schema synchronisation from routine deployments. The evaluate stage, also tag-triggered, builds and runs a dedicated evaluation-runner container inside the shared Docker network and saves `results.json` as a pipeline artefact, making each evaluation run independently version-traceable. The deploy stage transfers artefacts to the VM via SSH and `rsync`, restarts the core containers via Docker Compose and triggers a deploy-only Sync Container run to refresh the Ontology Agent pool. Each stage is change-triggered: a stage executes only when files within its monitored path have been modified, preventing unnecessary rebuilds and redeployments. Environment-specific secrets are managed through GitLab Secure Files and are never stored in the repository.

Comments The two-environment setup on a single VM keeps resource usage within the OST allocation while allowing development work to proceed in parallel with a stable production deployment. The change-triggered stage design means that a documentation-only commit does not trigger a container rebuild and an infrastructure-only change does not trigger the agent build pipeline. The primary operational constraint of this setup is that the Docker socket must be accessible on the VM for the Sync Container’s dynamic agent deployment to function correctly, which requires the deployment user to have the necessary Docker group membership on the OST VM.

- All evaluation results are produced under fixed, controlled hardware and network conditions, making cross-run and cross-model comparisons valid
- A single commit to `dev` or a merge to `main` is the only manual action required to deploy a new version to the corresponding environment
- Change-triggered CI stages prevent unnecessary rebuilds and reduce pipeline execution time for partial changes
- Secrets are never stored in the repository; they are injected at job time via GitLab Secure Files
- Two independent environments on the same VM allow development and production deployments to coexist without resource conflicts

5.7.2 ADR02: Predefined-Objects Single-Shot Payload Strategy

ADR ADR02
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture

Decision Record As established in Sections 5.4 and 5.5, any architecture that first creates the Atfinity case and only fills in the field values afterwards runs into the same problem. Once the case exists, there is no reliable way to match the party instances Atfinity created back to the individuals described in the Relationship Manager’s prompt. Three properties of Atfinity’s case model cause this. First, fields are addressed through composite keys rather than simple, stable identifiers. Second, Atfinity can automatically generate additional party instances for compliance reasons that were never explicitly requested in the prompt. Third, a single individual can hold multiple roles within the same case. Together, these make it unreliable to reconstruct, after the case has been created, which instance corresponds to which person. The decision was therefore made to assemble the complete `predefined_objects` payload before any call is made to Atfinity’s case creation endpoint. All field values for all parties are extracted by the multi-agent pipeline in advance and the assembled payload is submitted in a single atomic `POST /api/1/cases` request. Because all party data is embedded in the same request that generates the instances, no post-creation instance resolution step is required.

This approach is enabled by Atfinity’s `POST /api/1/cases` endpoint, which accepts a `predefined_objects` array in a single bulk creation request [3]. The first element of the array contains the fields of the outcome instance (for example, the account type) and all subsequent elements specify domain entities with their ontology key, role assignments and extracted field values, as illustrated in Figure 5.8.

```
{
  "process_id": 1,
  "predefined_objects": [
    {
      "account_type": "individual_account"
    },
    {
      "ontology": "Person",
      "roles": ["AccountHolder", "BeneficialOwner"],
      "first_name": "Test",
      "last_name": "San",
      "total_wealth": {
        "decimal_number": "2502050",
        "unit": "chf",
        "item": null
      },
      "source_of_wealth": [
        "company_sale",
        {"key": "other_free_text", "free_text_value": "Found it on the street"}
      ]
    }
  ],
  "preferred_language_key": "en"
}
```

Figure 5.8: Example `predefined_objects` payload for `POST /api/1/cases` [3]

Comments This decision is the architectural foundation of the entire bachelor thesis system. It enables multi-agent parallel extraction prior to any Atfinity API interaction and eliminates the instance-tracking problem that disqualified both the Iterative and Hybrid architectures from the study thesis. The primary trade-off is that the case is committed to Atfinity only after the complete extraction pipeline has finished, with no possibility of saving partial results incrementally. Validation errors returned by Atfinity on the

bulk POST reject the entire payload atomically, which necessitates the self-healing retry mechanism described in Section 5.7.9.

- Eliminates instance-identifier resolution for multi-party cases by embedding all party data in the case creation request
- Enables parallel multi-agent field extraction prior to any Atfinity API interaction
- Case creation is atomic: the case is either fully created or not created at all
- Field-level validation errors affect the complete payload and are addressed by a dedicated retry loop (see Section 5.7.5)

5.7.3 ADR03: LiteLLM as Provider-Agnostic LLM Interface

ADR ADR03
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR04, ADR05, ADR08

Decision Record Hard-coding a specific LLM provider and model into each agent’s source code would tie the system to a single vendor and turn every model change into a code modification and container rebuild. This would also make the bachelor thesis’s cross-model evaluation impractical, since comparing extraction quality, latency and cost across model configurations requires swapping the underlying LLM cleanly, without risking inadvertent behavioural differences introduced by the swap itself.

Three approaches were evaluated before selecting LiteLLM.

Custom per-provider adapters. Writing thin adapters over each provider’s native Python SDK was considered but rejected on the grounds of development time and ongoing maintenance. Provider abstraction is not a core contribution of this project and building a custom abstraction layer would divert effort from the extraction pipeline.

LangChain. LangChain is the most widely adopted Python framework for building LLM-powered applications and provides provider abstraction as one component of a comprehensive toolkit covering chains, agents, memory, retrieval and tool integration. That breadth is also the reason it was excluded. Introducing LangChain solely for provider routing imports a large dependency graph of abstractions that the system does not use, increases container image sizes, adds framework-layer indirection that complicates debugging and introduces framework-version upgrade risk. Since the extraction pipeline architecture was already complete, a full orchestration framework adds overhead without contributing value to this project.

The decision was made to give the system modular, swappable LLM backends: every agent-internal LLM call is routed through LiteLLM, a unified completion library that translates a standardised API call into the provider-specific format at runtime. Each agent group, for example all Ontology Agents, is bound to a dedicated environment variable holding a LiteLLM provider-prefixed model string such as `anthropic/claude-sonnet-4-5` or `openai/gpt-4o-mini`. Exchanging the LLM behind an entire agent group is therefore a configuration change, requiring only an environment variable update and a container restart, with no source code modification or image rebuild.

Comments LiteLLM normalises structural differences between provider APIs but not behavioural ones. Swapping a group’s model still requires validating prompt behaviour independently, particularly JSON output enforcement and instruction-following reliability, which vary between providers.

- Each agent group can be assigned the model that matches its reasoning requirements
- The LLM behind an agent group can be exchanged via a single environment variable, with no code changes or container rebuilds
- Makes the active model an explicit, inspectable configuration parameter, supporting systematic cross-model evaluation
- Provider-specific behavioural differences must still be validated independently for each model substitution

5.7.4 ADR04: Orchestrator as Central Coordination Component

ADR ADR04
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR01, ADR02

Decision Record The Multi-agent Preemptive Single-Shot architecture requires a component that receives the Relationship Manager’s request, coordinates all downstream agent invocations in the correct order, assembles the final payload and manages the submission to Atfinity. Three control topologies were evaluated for this role, as described in Chapter 4.

A decentralised peer-to-peer topology, in which agents communicate directly with each other, was rejected for two reasons. First, the pipeline has a natural sequential ordering constraint. Process determination must complete before routing can begin and routing must complete before extraction can start. Peer-to-peer messaging provides no mechanism to enforce this ordering without introducing coordination logic into every agent. Second, the MCP interface must be exposed as a single entry point that LibreChat can connect to; there is no natural candidate for that entry point in a flat peer network.

A hierarchical multi-tier topology, in which sub-coordinators manage groups of agents, was considered unnecessary given the current agent count. The three distinct agent types (Process Agent, Prompt Organizer Agent, Ontology Agents) do not form natural groups with internal coordination requirements; all coordination is between the top-level pipeline and individual agents.

The decision was therefore made to use a centralised Orchestrator as the sole coordination component. The Orchestrator is the only MCP endpoint exposed to LibreChat, the only component that calls Atfinity’s API directly and the single point through which all agent dispatch decisions, payload assembly and submission retry logic flow. This makes pipeline behaviour fully traceable, failure attribution unambiguous and the MCP tool surface clean.

Comments A centralised Orchestrator represents a single point of failure. In the present deployment, this is accepted. The OST VM runs a single stack per environment and onboarding requests are short-lived, so a restart recovers immediately. For a production deployment with high availability requirements, the Orchestrator would need to be replicated behind a load balancer.

- Provides a single, clean MCP endpoint that LibreChat can connect to without custom routing logic
- Enforces pipeline ordering constraints (process before routing, routing before extraction) without inter-agent negotiation
- Centralises all Atfinity API interactions, keeping agent containers free of external API dependencies
- Represents a single point of failure, accepted in the current deployment context

5.7.5 ADR05: Multi-Agent Decomposition with One Ontology Agent Per Ontology

ADR ADR05
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR01

Decision Record As established in Section 5.3, the Bank of Luxembourg’s CMS configuration exposes 23 ontologies and approximately 25 000 fields, a scale that makes passing the full schema to a single LLM prompt infeasible: it exceeds practical context window limits and simultaneously exposes the model to structurally similar fields across multiple ontologies, raising the risk of hallucination, incorrect field key selection and enumeration constraint violations. The decision was therefore made to decompose field extraction across multiple dedicated Ontology Agents, with exactly one agent deployed per ontology. Each agent receives only the schema for its assigned ontology and a focused sub-prompt containing the data for the specific entity it is extracting. This combination of bounded schema context and scoped input reduces the extraction problem to a manageable size for any individual agent and eliminates cross-ontology field confusion entirely.

Comments Agent scoping is the primary mechanism for maintaining extraction quality as the total schema size scales beyond what a single LLM can process reliably. An agent reasoning over a 300-field ontology schema operates in a fundamentally more controlled context than one reasoning over 25 000 fields simultaneously. The architectural consequence is that the number of active agent containers scales with the number of ontologies in the bank’s CMS configuration, which motivated the dynamic deployment mechanism described in Section 5.7.3.

- Reduces per-agent extraction context to one ontology schema, eliminating cross-ontology field confusion
- Extraction quality scales independently of the total CMS schema size
- A failure or incorrect extraction in one agent does not affect other agents’ results
- The number of agents scales with the CMS ontology count, requiring dynamic deployment at runtime (see Section 5.7.3)

5.7.6 ADR06: OntoDB as Shared Schema Persistence Layer

ADR ADR06
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR05, ADR07

Decision Record The multi-agent decomposition described in ADR05 requires every agent that participates in the pipeline to access ontology schemas: the Orchestrator needs the full ontology list to build its in-context directory and to inject the correct schema into each Ontology Agent; the Process Agent and Prompt Organizer Agent need the same directory to reason about process matching and entity routing; each Ontology Agent needs the schema for its assigned ontology as its extraction context. The Atfinity CMS is the single source of truth for these schemas. The question, therefore, is when and how that source-of-truth data should be made available to the components that consume it.

Three approaches were evaluated before arriving at the current solution.

Direct CMS API calls at inference time. Each component retrieves the schemas it needs from the Atfinity CMS API on every request. This eliminates any intermediate storage but creates a hard runtime dependency: every inference request blocks on CMS availability. In a parallel extraction phase where six Ontology Agent containers each issue a schema fetch simultaneously, the CMS becomes a per-request bottleneck. More critically, if the CMS is temporarily unavailable, the pipeline fails entirely, regardless of whether the schema data it would have returned has changed since the last request.

Schemas embedded in container images or environment variables. CMS schema data is baked into the agent images or injected as environment variables at deployment time. This removes the runtime CMS dependency but introduces a tighter one at deployment time. Any change to the bank's CMS configuration, whether adding an ontology, renaming a field or updating an enumeration, requires rebuilding and redeploying every affected container. This directly contradicts the self-configuring goal of ADR07, which is specifically designed to make the system adapt to CMS changes on the next startup without manual intervention.

A dedicated database service. Storing schemas in a relational or document database adds a queryable persistence layer but also adds a new service with its own availability, backup and upgrade requirements. The access pattern here is write-once-per-sync and read-many from local containers; there are no ad-hoc queries, no joins and no concurrency concerns. A database service provides infrastructure overhead without any benefit over direct file reads for this workload.

The decision was therefore made to persist the normalised CMS schema output of the Sync Container onto a dedicated Docker named volume, the OntoDB, immediately after each synchronisation cycle. The Sync Container is already responsible for fetching and normalising ontology data from the CMS; writing the result to a shared volume is a natural extension of that responsibility. All consuming services, the Orchestrator, the Process Agent, the Prompt Organizer Agent and all Ontology Agent containers, mount the volume read-only and read schemas directly from the YAML files without any network call or service adapter. Schema freshness is governed by the sync cycle: all components read the same snapshot written by the most recent synchronisation, ensuring consistent schema alignment across the entire pipeline.

Comments The OntoDB volume also serves as the coordination channel for dynamic deployment. The Sync Container writes an `agents_manifest.yaml` to the same volume after spawning all Ontology Agent containers, recording each agent's service URL. The Orchestrator reads this manifest at runtime to discover agent endpoints without hardcoded addresses, making the volume a single shared state location for both schema data and deployment metadata.

- Schema data is written once per sync cycle and read many times during inference, eliminating per-request CMS API calls
- All consuming containers access an identical schema snapshot, ensuring consistent alignment across the pipeline

- CMS unavailability during inference does not affect extraction; only the sync phase requires a live CMS connection
- Embedding schemas in images was rejected because it conflicts with the self-configuring deployment goal of ADR07
- A database service was rejected as over-engineering for a write-once, read-many, file-shaped workload

5.7.7 ADR07: Dynamic Ontology Agent Deployment via Docker Socket

ADR ADR07
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR02

Decision Record The multi-agent decomposition described in Section 5.7.2 requires one dedicated Ontology Agent container per ontology present in the connected bank’s CMS. The number of ontologies in any given Atfinity tenant is not fixed at design time and varies per bank. The Bank of Luxembourg configuration exposes 23 ontologies, but a different bank’s configuration may expose significantly more or fewer. Maintaining a static `docker-compose.yml` that enumerates every agent container by name would require manual updates whenever the bank adds, removes or renames an ontology in the CMS, creating a tight coupling between the deployment configuration and the specific tenant’s schema.

The decision was therefore made to implement dynamic agent deployment within the Sync Container via the Docker socket, as detailed in Section 5.6.4. On startup, the Sync Container discovers all available ontologies from the CMS, builds a shared agent image and launches one container per ontology. Service URLs are recorded in `agents_manifest.yaml`, which the Orchestrator reads at runtime without any hardcoded addresses.

Comments Dynamic deployment makes the system self-configuring with respect to the bank’s CMS schema. Adding or removing an ontology in Atfinity requires no changes to any deployment manifest or source code. The operational trade-off is that the Docker socket must be mounted into the Sync Container, granting it elevated privileges equivalent to administrative access on the Docker host. This is accepted as a controlled trade-off in the deployment environment. Docker’s layer caching ensures that a no-change image rebuild completes in seconds, making the additional startup overhead negligible in practice.

- System is self-configuring: ontology changes in the CMS are reflected automatically on the next startup
- No static agent enumeration is required in deployment configuration files
- The Orchestrator discovers agent service URLs at runtime via the manifest file, with no hardcoded service addresses
- Docker socket access grants elevated privileges to the Sync Container, which is accepted as a controlled trade-off

5.7.8 ADR08: Hybrid Sequential-Parallel Pipeline Execution

ADR ADR08

Author Arnel Veladzic

State Accepted

Related Multi-agent Preemptive Single-Shot Architecture, ADR01, ADR02, ADR03

Decision Record The pipeline comprises stages with fundamentally different dependency structures. The coordination stages, comprising prompt organisation, process determination, outcome extraction and ontology routing, form a strict sequential dependency chain. Each stage’s output is a direct prerequisite for the next. The extraction stage, by contrast, consists of independently scoped Ontology Agent calls with no dependency on each other’s results.

A fully sequential execution pattern was rejected on latency grounds. In the Bank of Luxembourg configuration, a single Person ontology agent invocation can take between 30 and 60 seconds due to the size of the schema. For a trust case requiring six simultaneous agent invocations, running these calls one after another would be expected to push the extraction phase alone past the NFR-1 threshold before any Atfinity submission is attempted.

A fully parallel execution pattern from the start is structurally impossible. The Ontology Agents require the organised prompt and the per-entity sub-prompts as their input, neither of which exists until the coordination stages have completed.

The decision was therefore made to apply a hybrid pattern. The coordination stages execute sequentially. Once all routing outputs are available, the extraction stage fans out to all required Ontology Agents concurrently. Each agent call is independent, since there is no shared state and no ordering constraint between individual agents, which makes full concurrency safe. The total extraction latency is bounded by the slowest individual agent rather than the cumulative sum. Assembly and submission then execute sequentially once all extraction results are collected. This pattern concentrates the parallelism precisely where it delivers the greatest latency benefit, while respecting the ordering constraints of the coordination stages.

Comments The coordination stages are individually fast and contribute only a few seconds to total latency. The dominant cost is the extraction stage, which is why concentrating the parallelism there has the greatest impact on end-to-end latency. Parallel dispatch is safe because Section 5.6.6 established that all agent inputs are fully assembled before the fan-out begins. A failure in any individual extraction agent is isolated: it returns an empty field set without cancelling or delaying the remaining agents.

- Coordination stages execute sequentially, enforcing data dependency constraints without inter-agent synchronisation
- Parallel extraction bounds total latency to the slowest individual agent, not the cumulative sum
- Parallel dispatch is safe because all agent inputs are fully determined before any call is made (Section 5.6.6)
- A failure in one extraction agent does not block or cancel the remaining agents in the fan-out
- Total pipeline latency remains within NFR-1 (≤ 120 s) even for the most complex multi-party cases evaluated

5.7.9 ADR09: Self-Healing Retry Loop for Atfinity Validation Errors

ADR ADR09
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR01

Decision Record Atfinity’s `POST /api/1/cases` endpoint applies strict schema validation to every field in the submitted payload. The endpoint is atomic: if any field fails validation, the entire request is rejected with HTTP 400 and a per-field error response. No partial case is created. The Bank of Luxembourg configuration enforces strict enumeration constraints, structured value formats for monetary amounts, multi-value array types and field-level business rules. Given the complexity and scale of this schema, some first-attempt validation errors are expected even when individual agent extraction quality is high. Surfacing these raw Atfinity validation errors to the Relationship Manager and requiring manual data correction would effectively delegate schema compliance to the user, which is not acceptable.

The decision was made to implement a self-healing retry loop within the Orchestrator with a maximum of five attempts. On each rejection, the Orchestrator classifies each failed field into one of two categories. Structural errors, where a value is technically malformed regardless of its content (for example, a list submitted as a string literal with bracket notation), are patched directly by the Orchestrator without an additional LLM invocation, since the correction is a deterministic reformatting operation. Semantic errors, where a value is syntactically valid but incorrect in content (for example, an unrecognised enumeration key), are routed to the responsible Ontology Agent’s `POST /correct` endpoint, as described in Section 5.6.4. If a correction attempt fails or the responsible agent cannot resolve the field, the field is stripped from the payload before the next retry, trading extraction completeness for a higher chance that the remaining payload passes validation. Stripping does not guarantee success: if the payload still fails validation after the maximum number of attempts, the Orchestrator reports a permanent failure to the Relationship Manager rather than submitting an incomplete case.

Comments The classification of errors into structural and semantic categories is essential because each type requires a different resolution strategy. Routing a semantic error to an inline patch or routing a structural error to an LLM correction call, would produce incorrect results in both cases. The five-retry ceiling prevents the loop from running indefinitely on validation constraints that cannot be resolved within the available schema information. Stripping an uncorrectable field reduces extraction completeness for that case but increases the likelihood that the next retry attempt passes Atfinity’s validation. It is not a guarantee of success. A case that still fails validation after the maximum number of attempts is reported to the Relationship Manager as a permanent failure rather than being submitted incomplete.

- Shields the Relationship Manager from Atfinity’s strict validation model by resolving errors automatically
- Structural errors are resolved inline by the Orchestrator without LLM involvement
- Semantic errors are resolved by routing the rejection context back to the responsible Ontology Agent for LLM-assisted correction
- Uncorrectable fields are stripped from the payload between retries to improve the odds that the next attempt passes validation, without guaranteeing that the case will ultimately be created
- The five-retry limit prevents an infinite loop on unresolvable validation conflicts

5.7.10 ADR10: Schema Learning via SchemaDB

ADR ADR10
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR01

Decision Record Atfinity’s `POST /api/1/cases` endpoint requires a specific top-level payload structure that includes fields not present in the ontology schema files retrieved from the CMS. The most critical of these is `preferred_language_key`, which determines the language in which Atfinity renders the case data. This field is not discoverable through any schema API endpoint; its expected value must be established from a successfully accepted submission. Additional structural conventions, such as the positioning of the outcome instance object at the top of the `predefined_objects` array and the format of optional metadata fields, are not fully documented for the specific tenant configuration in use.

Hardcoding these structural defaults directly in the Orchestrator would create a maintenance dependency. Any change to Atfinity’s expected payload structure for the connected tenant would require a code modification and a container redeployment. Deriving them from documentation alone is unreliable given that Atfinity’s API behaviour is partially tenant-configurable and may differ from the published specification.

The decision was therefore made to implement a persistent payload learning mechanism in the Orchestrator, as described in Section 5.6.4. After each successful Atfinity submission, the accepted payload’s structural skeleton is persisted and used as the initialisation basis for subsequent payload assembly. Structural defaults such as `preferred_language_key: en` are carried forward automatically without any code-level configuration.

Comments The first successful submission bootstraps the learning mechanism. The Orchestrator ships with a minimal default skeleton that contains the required structural shape and the base schema is refined from that first accepted payload onwards. The process identifier and the `predefined_objects` array are always overwritten unconditionally during payload assembly, so the structural skeleton contributes only the top-level fields that remain consistent across submissions. The archived payloads directory provides a complete audit trail of all accepted submissions, independently useful for debugging and post-hoc validation.

- Structural payload defaults are learned from accepted submissions rather than hardcoded, removing a maintenance dependency on tenant-specific API behaviour
- `preferred_language_key` and other non-schema structural fields are automatically preserved across submissions without code-level configuration
- Every accepted payload is archived with a timestamped filename, providing a complete audit trail
- The learning mechanism is self-bootstrapping and activates from the first successful submission without manual initialisation

5.7.11 ADR11: Two-Stage Prompt Scoping for Multi-Party Extraction

ADR ADR11
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR02, ADR12

Decision Record Multi-party onboarding prompts describe several individuals in a single unstructured text. Routing this combined input directly to Ontology Agents produces two systematic failure modes. The first is name-based misattribution. When two individuals share the same first name, for example Peter MARBURGER and Peter NIEDEREGGER, the extraction LLM may assign the date of birth, nationality or contact details of one person to the other. The second is shared-attribute omission. Attributes such as investment horizon, risk profile and MiFID questionnaire answers are typically stated once and apply to all parties jointly. A naive per-entity split would strip these from some individuals' sub-prompts, causing required fields to be missed despite being present in the original input.

The decision was made to address both failure modes through a two-stage input transformation applied before any Ontology Agent is invoked.

In the first stage, a dedicated Prompt Organizer Agent reorganises the raw scattered input into clearly labelled, entity-separated sections. Each section groups all data attributable to one individual under a named header. Shared contextual attributes are explicitly duplicated into every relevant section. The agent never adds, infers or removes content; its sole responsibility is restructuring. It is deployed as an independent microservice rather than inline within the Orchestrator so that prompt reorganisation remains decoupled from pipeline control flow and can be assigned a smaller, cheaper model. If its LLM call fails, the agent returns the original prompt unchanged and the pipeline continues without interruption.

In the second stage, the Orchestrator's routing step decomposes the organised text into per-entity sub-prompts, one per identified individual. Each sub-prompt contains only the data attributable to that specific person plus all shared contextual attributes. The routing LLM verifies role attribution by full name against the source text, preventing misattribution between individuals who share a first name. These scoped sub-prompts are then dispatched to the Ontology Agents in the subsequent parallel fan-out.

Comments Both stages are necessary and complementary. The first stage resolves the interleaving problem, grouping scattered data so the second stage has clean, entity-separated input to work with. The second stage resolves the contamination problem at the agent boundary, ensuring each Ontology Agent receives only its assigned entity's data. Skipping either stage reintroduces the failure modes this ADR is designed to eliminate. The two stages may appear to duplicate each other since both group the prompt by entity; Section 5.6.6 (Preview Step 4) draws out why they are not redundant: the first stage is a purely textual regrouping with no knowledge of Atfinity's data model, while the second stage maps each grouped entity onto a concrete ontology type, instance count and role using the Orchestrator's MemOrch context, a mapping the first stage is structurally unable to make.

- Eliminates cross-entity data contamination before any extraction agent is invoked
- Shared contextual attributes are explicitly included in every relevant sub-prompt, preventing omission
- The Prompt Organizer is an independent service, enabling a lower-cost model assignment for this low-complexity task
- Role attribution is verified by full name in the source text, preventing misattribution between individuals with shared first names
- Graceful fallback to the original prompt prevents preprocessing failures from blocking the pipeline

5.7.12 ADR12: Preview Approval Gate with Pipeline Result Caching

ADR	ADR12
Author	Arnel Veladzic
State	Accepted
Related	Multi-agent Preemptive Single-Shot Architecture

Decision Record Across the iterative development and evaluation of the extraction pipeline, a consistent pattern emerged: the quality of the extracted result is bound by the quality of the input prompt it is derived from. The better-structured and more complete the Relationship Manager’s description of a case, the more accurate the resulting field extraction and process match. At the same time, RMs communicate in free-form natural language during a live conversation and cannot reasonably be expected to supply perfectly structured, unambiguous descriptions of every party and field on the first attempt. Left unaddressed, this gap between realistic input quality and the pipeline’s sensitivity to it would translate directly into incorrect case submissions. A wrong initial submission is not unrecoverable: a Relationship Manager can correct it afterwards, either manually through Atfinity’s CMS interface or conversationally through the system’s own correction tools, such as `update_case_field`, `add_case_instance` and `delete_case_instance`, which together allow virtually every aspect of a case to be amended or removed after creation. However, every such correction is pure operational overhead: additional manual edits in the CMS or additional conversational turns and LLM calls to fix what should have been right the first time. The goal of the approval gate is therefore not to guard against an irreversible action, but to minimise both the likelihood of an incorrect initial submission and the corrective effort it would otherwise generate downstream.

To close this gap before any data reaches Atfinity, the decision was made to introduce a dedicated `preview_onboarding` MCP tool that runs the prompt reorganisation, process determination, outcome ontology extraction and ontology routing steps of the pipeline (Preview Steps 1 through 4, Section 5.6.6), but stops before the parallel field extraction of the remaining entities, payload assembly and Atfinity’s case creation endpoint. The tool returns a structured summary to the Relationship Manager for review, covering the entities identified from the prompt with their assigned roles, the onboarding process matched to the case and the extracted outcome ontology. This gives the RM a concrete, reviewable checkpoint to catch and correct misextractions stemming from an imperfect prompt before they are committed, getting the case right on the first submission rather than relying on after-the-fact corrections. The RM can then confirm by calling `create_client_onboarding` or abort without any case being created.

The critical design question for this workflow was whether `create_client_onboarding` should re-run the full extraction pipeline from scratch on confirmation. Re-running introduces two problems. First, it doubles the LLM call count and the associated latency and cost. Second, LLM-based extraction is non-deterministic: re-running the same prompt may produce different field values, meaning the data the RM approved in the preview may not match what is actually submitted to Atfinity.

To avoid both problems, all intermediate pipeline results produced by `preview_onboarding` are stored in an in-memory cache: the reorganised prompt, the process determination result, the outcome ontology extraction and the list of identified ontologies with their per-instance sub-prompts. When `create_client_onboarding` is called, it reads all cached results and skips the corresponding pipeline steps (prompt reorganisation, process determination, outcome extraction and ontology routing) entirely. It then proceeds to the parallel field extraction of the identified entities using the cached per-instance sub-prompts the RM reviewed, before assembling the payload and submitting it to Atfinity. The cache uses two lookup mechanisms: a pending slot that always holds the most recent preview and is consumed without prompt-string matching and a SHA-256 hash-keyed fallback for cases where the pending slot was cleared by a container restart between the preview and the confirmation call.

Comments The pending slot does not rely on matching prompt strings, so it still works even if Claude rephrases the Relationship Manager’s wording between the preview and confirmation calls. The hash-keyed fallback covers container restart scenarios. The cache is in-memory and does not survive container restarts, which is accepted as a minor edge case given that the time between a preview and its confirmation is typically short. Direct calls to `create_client_onboarding` without a prior preview bypass the cache and execute the full pipeline immediately.

- Provides a review-before-commit workflow that lets the RM catch and correct misextractions caused by imperfect natural-language input before they reach Atfinity, minimising both incorrect initial submissions and the corrective effort they would otherwise create downstream
- Cached pipeline results guarantee that the data submitted to Atfinity is identical to what the RM reviewed in the preview
- Eliminates redundant LLM calls on confirmation, reducing cost and latency
- Cache is automatically evicted after a successful case creation to prevent stale data from affecting subsequent requests

5.7.13 ADR13: LibreChat as Frontend and MCP Client

ADR ADR13
Author Arnel Veladzic
State Accepted
Related Multi-agent Preemptive Single-Shot Architecture, ADR12

Decision Record *Note: LibreChat is not a new decision for the bachelor thesis. It was already selected and validated as the frontend and MCP client in the preceding study thesis and is retained here unchanged. The study thesis baseline decisions are summarised in Section 5.3. This ADR documents the choice for completeness and records the alternatives that were evaluated against it.*

The system requires a web-based conversational interface through which the Relationship Manager can interact with the multi-agent pipeline in natural language and an MCP-compatible client that can relay the LLM’s tool invocations to the Orchestrator’s MCP endpoint. Three options were evaluated.

OpenWebUI is a popular open-source chat interface for self-hosted LLMs. At the time of the study thesis, OpenWebUI did not support the Anthropic API natively, making it incompatible with the Claude model family. This was the reason it was not selected then. Since LibreChat had already been validated in the study thesis, OpenWebUI was not reconsidered for the bachelor thesis.

A self-developed UI would allow full control over the interface and the MCP integration but was rejected on scope grounds. Frontend development is not the research contribution of this thesis; building a custom chat application from scratch would consume significant sprint capacity needed for the multi-agent architecture itself. The system’s value lies in the backend pipeline, not the UI layer.

LibreChat provides native support for the Anthropic API, built-in MCP client capability, a familiar conversational interface and full Docker containerizability. It was validated in the preceding study thesis, making it a proven choice that carries forward established deployment patterns. The decision was therefore made to retain LibreChat for the bachelor thesis.

Comments LibreChat is self-hosted, meaning no conversation data is transmitted to a third-party chat service. This is directly relevant to the banking context, where client data transmitted through the chat interface must not leave the controlled deployment environment.

- Native Anthropic API support enables the full Claude model family without custom adapters
- Built-in MCP client support connects Claude’s tool-calling capability directly to the Orchestrator’s MCP endpoint
- Full containerisability preserves the Docker Compose deployment pattern validated in the study thesis
- Self-hosted deployment ensures that conversation content remains within the controlled OST VM environment
- OpenWebUI was excluded due to missing Anthropic API support at the time of the study thesis; LibreChat was retained as the already-validated choice
- A self-developed UI was excluded as out of scope; frontend development does not contribute to the research goals

5.7.14 ADR14: Caddy as Reverse Proxy

ADR ADR14
Author Arnel Veladzic
State Accepted
Related Deployment Environment, ADR11

Decision Record The LibreChat frontend must be served over HTTPS, since the banking-context application transmits client data through the browser interface. A reverse proxy capable of TLS termination is therefore required at the edge of the Docker network. Three alternatives were evaluated.

nginx is the most widely deployed reverse proxy and is well-suited to static configuration. However, TLS certificate provisioning and renewal with Let's Encrypt requires manual integration of certbot or equivalent tooling, adding operational overhead. Certificate renewal must also be scheduled explicitly and monitored to prevent expiry.

Traefik supports automatic Let's Encrypt certificate management via Docker label-based configuration. For the two-environment setup on the same host, Traefik's dynamic routing model introduces additional configuration complexity. Routing rules, middleware chains and certificate resolvers must be explicitly defined per service and the interaction between the production and development containers on a shared network requires careful label scoping.

Caddy provisions and renews TLS certificates from Let's Encrypt automatically via the ACME protocol at startup, with no manual configuration required. Subdomain-based routing between the production environment (`mas-preemptive.asoftware.ch`) and the development environment (`mas-preemptive-dev.asoftware.ch`) is expressed in a concise Caddyfile. Certificate renewal is handled entirely by Caddy's internal scheduler. The decision was therefore made to use Caddy as the shared reverse proxy for both deployment environments.

Comments Caddy is deployed once and shared across both the production and development environments, as described in Section 5.6.2. Because certificate management is automatic, no operational intervention is required when certificates approach expiry, which is particularly important for an evaluation environment that runs unattended between sprint iterations.

- Automatic TLS certificate provisioning and renewal via the ACME protocol eliminates certificate management overhead
- Subdomain-based routing between production and development environments is expressed concisely without per-service middleware chains
- A single shared Caddy instance covers both environments, reducing infrastructure duplication on the OST VM
- nginx was excluded due to manual certificate renewal requirements; Traefik was excluded due to configuration complexity for the two-environment setup

Chapter 6

Implementation

This chapter describes selected implementation aspects of the Multi-agent Preemptive Single-Shot architecture. Section 6.1 lists the key frameworks and libraries the system depends on. Section 6.2 explains how the synchronisation container populates the shared schema volume. Section 6.3 describes the mtime-based auto-refresh pattern shared by all in-memory context stores. Section 6.4 covers the provider-agnostic LLM integration layer. Section 6.5 details the schema chunking and partial JSON recovery mechanisms that make reliable extraction of large schemas possible. Section 6.6 explains the dual-slot preview cache. Section 6.7 documents the error classification and correction routing of the self-healing submission loop. Section 6.8 describes the payload-skeleton learning mechanism. Section 6.9 covers the programmatic Docker container deployment for ontology agents. Section 6.10 documents the GitLab CI/CD pipeline, covering secret injection, deploy-stage sequencing and the tag-gated evaluation workflow. Section 6.11 details the deterministic evaluation framework, including the test suite composition, the ground-truth-driven scoring engine, the token cost tracking infrastructure and the retry count accumulation mechanism. Section 6.11.5 discusses the scope and extension path for isolated individual agent evaluation.

6.1 Technology Stack and Dependencies

The system is implemented entirely in Python 3.12 and containerised with Docker. The following table lists the key libraries and frameworks used across the agent containers.

Library / Framework	Role in the system
FastMCP	MCP server framework used by the Orchestrator to expose tools over the <code>streamable-http</code> transport.
FastAPI	HTTP server framework used by the Process Agent, Prompt Organizer Agent and Ontology Agent containers to expose REST endpoints.
LiteLLM	Provider-agnostic LLM interface used by all agents; abstracts over OpenAI, Anthropic and other compatible providers via a unified API.
docker (SDK)	Python Docker SDK used by the sync container to build images and manage agent container lifecycles at runtime.
PyYAML	YAML parsing and serialisation for all schema, process and agent configuration files written by the sync container.
requests	Synchronous HTTP client used by the Orchestrator for inter-container communication with agent services.

Table 6.1: Key Python libraries used in the Multi-agent Preemptive Single-Shot architecture

The user-facing frontend is provided by LibreChat, an open-source chat application configured to connect to the Orchestrator’s MCP endpoint. Conversation history and user session data are persisted in a MongoDB instance managed by Docker Compose. The deployment infrastructure uses GitLab CI/CD with SSH-based deployment to the OST virtual machine, as described in Section 5.7.1. All environment-specific configuration is supplied via `.env.dev` and `.env.prod` files and injected into containers at runtime by Docker Compose.

6.2 Schema Synchronisation

The synchronisation container is the entry point through which all live Atfinity schema data enters the system. On startup it runs `sync_runner.sh`, which launches `ontology_mapping.py` and `processes_mapping.py` as background processes in parallel. Both scripts authenticate against Atfinity’s REST API and write their outputs to the shared `onto-db` Docker volume, which is mounted read-only in all agent containers. Once both scripts have completed, the container exits. Schema re-synchronisation requires an explicit re-trigger via a tagged CI/CD release or a manual container re-run; it does not happen automatically on a schedule. This decision was deliberate: periodic synchronisation would place unnecessary load on the Atfinity sandbox API and introduce operational noise during evaluation runs, while the Bank of Luxembourg’s configuration remained stable throughout the thesis. In a production context, a scheduled cron-based restart would be the natural extension.

```
# sync_runner.sh (simplified)
python3 ontology_mapping.py &
ONTO_PID=$!
python3 processes_mapping.py &
PROC_PID=$!
wait $ONTO_PID; wait $PROC_PID
```

The `ontology_mapping.py` script produces two types of output per run. For each ontology it writes a full YAML schema file including all field keys, display names, types and enumerated possible values. Alongside these it writes a lightweight `orch_context.yaml` file containing only the ontology key, display name, description and available roles, which is the minimal context the routing LLM needs without loading any full schema. To avoid overloading Atfinity’s API, the script caps concurrent requests at six API calls and five AI-enrichment calls simultaneously, with automatic retry on HTTP 429 responses. The `processes_mapping.py` script writes `processes/processes_mapping.yaml`, which contains one entry per Atfinity process with its numeric ID, display name, description, key differentiator for disambiguation, LLM-generated example intent phrases and the outcome ontology reference. The intent phrases are the primary matching signal used by the Process Agent at inference time. Figure 6.1 shows the full directory structure written to the shared volume after a completed synchronisation run.

```
onto-db/
  orch_context.yaml
  ontologies/
    ontology_Account.yaml
    ontology_Person.yaml
    ...
  processes/
    processes_mapping.yaml
  agents/
    agent_Account.yaml
    agent_Person.yaml
    agents_manifest.yaml
  schema-db/
    base_schema.json
    successful_payloads/
```

Figure 6.1: Directory structure of the `onto-db` shared Docker volume after initial synchronisation

6.3 In-Memory Context Stores

Rather than reading YAML files from disk on every request, each agent holds its context in a small in-memory store that auto-refreshes whenever the sync container writes a newer version. This pattern eliminates file I/O on the hot path while ensuring that schema and process updates written by the Sync Container propagate to all agents without a container restart. The implementation uses a lazy modification-time (mtime) based check. Before every read the store calls `stat()` on the file and compares

the modification time against the value recorded at last load. A reload only happens when the sync container has actually written a new version.

```
# Shared mtime-refresh pattern (simplified)
def _refresh_if_stale(self):
    if self._filepath.stat().st_mtime > self._loaded_mtime:
        self._load()

def all(self):
    self._refresh_if_stale()
    return list(self._entries)
```

Four components apply this pattern. `MemOrch` (implemented in `mem_orch.py`) loads `orch_context.yaml` and exposes the ontology directory to the Orchestrator’s routing LLM. `MemPDA` (`process_agent.py`) loads `processes_mapping.yaml` and provides the process list to the Process Agent before each determination call. The `PromptOrganizer` (`prompt_organizer.py`) loads `orch_context.yaml` independently to obtain the list of available entity types, which it passes to the LLM so that section headers in the reorganised prompt match the actual ontology keys of the connected bank’s configuration. Each `OntologyAgent` instance applies the same check to its own full schema file at `onto-db/ontologies/ontology-{key}.yaml`, so schema updates reach running extraction agents without a container restart. No locking is needed in any case because the sync container writes files atomically and the in-memory list is replaced as a whole on reload.

6.4 LLM Integration via LiteLLM

All LLM calls across the system go through LiteLLM, a thin library that presents a single unified completion interface regardless of the underlying provider. This abstraction means the entire model configuration can be changed by editing environment variables without touching any agent code. Each agent role exposes a dedicated variable so that different models can be assigned to different roles at deployment time.

```
# Model assignment per agent role (.env files)
ORCHESTRATOR_LLM_MODEL=anthropic/claude-sonnet-4-6
PROCESS_AGENT_LLM_MODEL=openai/gpt-4o-mini
ONTOLOGY_LLM_MODEL=anthropic/claude-sonnet-4-6
PROMPT_ORGANIZER_LLM_MODEL=openai/gpt-4o-mini
```

Inside each agent the model string is read from the environment at call time via `os.environ.get("LLM_MODEL")`, so a rolling deployment that changes a model takes effect on the next container start without rebuilding any image. LiteLLM resolves the provider prefix (for example `anthropic/` or `openai/`) to the correct API client and authentication headers automatically. This design is the technical prerequisite for RQ3 (Section 7.7.3), which evaluates whether assigning smaller, cheaper models to lower-complexity roles yields measurable accuracy or cost advantages over a uniform-model configuration.

One temperature distinction is worth noting for reproducibility. The Prompt Organizer Agent runs at `temperature=0.0`, making its restructuring output fully deterministic. The same scattered prompt always produces the same organised sections. All Ontology Agents run at `temperature=0.1`, a near-deterministic setting that introduces a small amount of stochastic variance. This is intentional. A strictly zero temperature would make the model always select the single highest-probability token at each step, deterministically locking in that choice even when a competing interpretation is nearly as likely. On genuinely ambiguous fields, this means a marginally favoured but incorrect token would be produced on every run with no chance of correction, whereas a small amount of sampling at $T = 0.1$ allows the near-tied alternative to surface in at least some runs. The resulting residual variance is exactly what the stochastic repetition protocol in the evaluation framework (Section 6.11.4) is designed to measure and report.

6.5 Schema-Aware Field Extraction

Each ontology agent extracts structured field data from a natural-language prompt by presenting the prompt alongside its full ontology schema to the LLM and requesting a flat JSON object mapping field

keys to extracted values. Two technical challenges shaped this component: the context-length limits of smaller models when processing large schemas and the response-truncation behaviour that occurs when the model reaches the output token budget.

6.5.1 Schema Chunking

The Person ontology in the OST Atfinity deployment contains more than 130,000 tokens worth of schema content, exceeding the context window of most models when combined with the prompt and system instructions. The solution splits the schema's `informations` list into fixed-size windows of `SCHEMA_CHUNK_SIZE = 30` fields each. Every chunk receives the full schema metadata (ontology key, description, roles) for context; only the `informations` slice differs per chunk. All chunks are dispatched in parallel using a `ThreadPoolExecutor` and their results are merged by field key.

```
# Chunk split and parallel dispatch (simplified)
chunks = [
    {**schema, "informations": informations[i:i + CHUNK_SIZE]}
    for i in range(0, len(informations), CHUNK_SIZE)
]
with ThreadPoolExecutor(max_workers=len(chunks)) as ex:
    futures = {ex.submit(process_chunk, prompt, c): i
                for i, c in enumerate(chunks)}
    for f in as_completed(futures):
        merged.update(f.result())
```

Each chunk call is bounded by `LLM_CHUNK_TIMEOUT = 120` seconds and `LLM_MAX_TOKENS = 4096` output tokens. The timeout prevents a single stalled request from blocking the parallel dispatch indefinitely, since all chunks are awaited together via `ThreadPoolExecutor` and one hung call would otherwise stall the entire extraction. The output token cap reflects that each chunk only needs to return a JSON object for at most `SCHEMA_CHUNK_SIZE = 30` fields, keeping generation cost and latency predictable across the parallel calls; however, for unusually large chunks this same bound can cause the model's output to be cut off mid-object, a failure mode handled by the Partial JSON Recovery mechanism described in Section 6.5.2. Two validation steps are applied before merging chunk results. First, field keys returned by the model that are not present in the schema chunk are classified as hallucinations and dropped, preventing the agent from inventing field identifiers not in its assigned schema. Second, fields whose extracted value is `null` are also dropped before the merge (`non_null = {k: v for k, v in chunk_result.items() if v is not None}`). This means an agent that returns a `null` for a field is treated identically to an agent that omits it entirely. Neither contributes a value to the final field-value dict. This design avoids sending explicit null values to Atfinity's API, which rejects null-valued fields as malformed rather than treating them as empty.

6.5.2 Partial JSON Recovery

When an LLM response is cut off at the token limit the JSON object is truncated mid-entry, producing an unparseable string. A recovery function strips the incomplete trailing entry and closes the object, salvaging all field pairs that were fully generated before the cutoff.

```
# Partial JSON recovery (simplified)
def _recover_partial_json(raw: str) -> dict:
    stripped = raw.rstrip()
    last_comma = stripped.rfind(",")
    if last_comma == -1:
        return {}
    candidate = stripped[:last_comma] + "\n}"
    try:
        return json.loads(candidate)
    except json.JSONDecodeError:
        return {}
```

This mechanism was introduced to recover from the truncation risk inherent in the `LLM_MAX_TOKENS` bound described in Section 6.5.1, after observing Person schema chunk responses exceeding 46,000 characters in

practice, where the model generated more output than the API endpoint returned, producing a mid-string truncation. Recovery ensures the majority of a chunk’s extraction is preserved even in the worst-case scenario.

6.6 Preview Cache

When the Relationship Manager requests a preview before approving a case creation, the preview results must be reusable by the subsequent `create_client_onboarding` call without repeating service and LLM calls already made. Keying the cache by the prompt string is insufficient because the approval call may differ slightly in phrasing from the preview call, producing a mismatch.

The `preview.cache` module solves this with a dual-slot design. A module-level `_pending` slot always stores the most recent preview result regardless of prompt text. The execution tool queries this slot first using `get_pending()`, which requires no prompt argument and is immune to phrasing drift between the preview and approval calls. A second SHA-256-keyed `_store` dictionary serves as a fallback for cases where the pending slot was cleared by a container restart between the two calls.

```
# Dual-slot cache (simplified)
_store: dict = {}    # SHA-256 keyed fallback
_pending = None      # always holds the latest preview result

def put(prompt, result):
    global _pending
    _store[sha256(prompt)] = result
    _pending = result    # no prompt matching needed on retrieval

def get_pending():
    return _pending
```

Each cache entry stores four fields: the `organized_prompt` produced by the Prompt Organizer Agent, the `process_match` returned by the Process Agent (including the process ID, display name and outcome ontology), the `outcome_extraction` produced by the outcome ontology agent and the `identified_ontologies` list with per-instance sub-prompts and role assignments from the routing step. When `create_client_onboarding` finds all four entries in the cache, it skips the Prompt Organizer, Process Agent, routing LLM and outcome agent calls entirely, proceeding directly to the parallel Ontology Agent fan-out.

A critical correctness property is enforced via the `is_correction` flag on `preview_onboarding`. When `is_correction=False` (the default for any first call in a conversation), the pending slot is deliberately not consulted. This prevents a stale preview from a previous conversation from being silently reused as the starting point for a new request. The pending slot is only read when `is_correction=True`, which the LLM sets only when it has explicitly confirmed that the RM is correcting a preview shown earlier in the same conversation.

Both slots are evicted atomically after a successful case creation to prevent stale data from affecting subsequent requests.

6.7 Self-Healing Submission Loop

The `POST /api/1/cases` endpoint is atomic. Atfinity either accepts the full payload or rejects it entirely with a structured error identifying the offending object or field. The Orchestrator applies automated corrections and retries up to `MAX_RETRIES = 5` times rather than surfacing raw API errors to the user.

On each rejection, every error returned by Atfinity is classified into one of three categories before any repair is attempted.

Ontology-level errors arise when an entire object in the `predefined_objects` array violates a structural constraint at the case level, for example when the payload contains a duplicate outcome instance. These are identified by Atfinity returning an error referencing the ontology key rather than a specific

field. The Orchestrator removes the entire offending object from `predefined_objects` before the next retry.

Structural field errors occur when a field value is technically malformed regardless of its semantic content, typically because the LLM formatted a value as a string with bracket notation instead of a proper JSON array. These are detected by `_is_structural_error()`, which applies a compiled regular expression to the Atfinity error message looking for keywords such as `bracket`, `parenthes`, `syntax`, `parse`, `json`, `format`, `malformed`, `invalid format`, `unexpected token` or `decod`.

```
_STRUCTURAL_ERROR_RE = re.compile(
    r"\b(bracket|parenthes|syntax|parse|json|format|malformed|"
    r"invalid\s+format|unexpected\s+token|decod)\b",
    re.IGNORECASE,
)
```

Structural errors are repaired inline by the Orchestrator using `ast.literal_eval()` to reparse malformed string values. If the reparse fails the field is stripped.

Semantic field errors occur when a value is syntactically valid but incorrect in content, for example an unrecognised enumeration key or a business rule violation. These are routed back to the responsible ontology agent container via `POST /correct`. Multiple affected agents are contacted in parallel via a second `ThreadPoolExecutor` fan-out. Each agent receives the original prompt, the rejected field key, the submitted value and the Atfinity error message and returns a corrected value or `null`. If an agent is unreachable the field is stripped as a fallback.

If Atfinity returns a rejection that contains no identifiable field or ontology reference, the loop breaks immediately. Resending the same payload cannot succeed and the Orchestrator reports a permanent failure to the Relationship Manager.

6.8 Schema Learning

Every payload that Atfinity accepts is archived and used to update the skeleton that future payloads are assembled from. After each successful submission the Orchestrator archives the accepted payload to `schema-db/successful_payloads/{timestamp}_{process_id}.json`, providing a full audit trail of accepted field combinations.

The `_update_base_schema()` function then reads the current `base_schema.json` and rebuilds its `predefined_objects` list from the newly accepted payload. All field keys are retained and all values are cleared to `null`, producing a structural template for the next assembly. Critically, the update preserves the top-level structural fields that sit outside `predefined_objects`, including `preferred_language_key: "en"`, `owner_email` and `name`, which are required by the specific tenant configuration but are not discoverable through Atfinity's schema API.

The skeleton also carries a `_meta` bookkeeping block containing a `version` counter, a `successful_submissions` count, a `last_updated` timestamp and an embedded copy of the Atfinity API documentation and example curl command. This makes the stored skeleton self-documenting and independently useful for debugging. Before any payload is sent to Atfinity, the Orchestrator calls `strip_meta()` to remove all keys prefixed with `_meta`, ensuring that no internal bookkeeping fields reach the API.

On first invocation `get_base_schema()` self-bootstraps by writing a default skeleton to disk before returning it. Because the skeleton is consulted at the start of every payload assembly, each successful submission makes the next assembly start from a structurally more accurate baseline.

6.9 Dynamic Agent Deployment

Ontology agent containers are not defined statically in `docker-compose.yml`. Instead they are provisioned programmatically at runtime by `sync/deploy_agents.py` using the Python Docker SDK, driven entirely by the live `orch_context.yaml` produced by the sync step. The number of deployed agents automatically matches the number of ontologies in the active Atfinity instance.

Before launching containers the script introspects its own running container to resolve two parameters dynamically: the exact Docker volume name mounted at `ONTO_DB_PATH` and the overlay network the sync container is attached to. Both values are obtained from the Docker daemon via the SDK, eliminating hardcoded names.

```
# Volume and network discovery (simplified)
own = client.containers.get(socket.gethostname())
for mount in own.attrs["Mounts"]:
    if mount["Destination"] == ONTO_DB_PATH:
        volume_name = mount["Name"]
networks = own.attrs["NetworkSettings"]["Networks"]
network = next(n for n in networks
               if n not in ("bridge", "host", "none"))
```

The agent image is always rebuilt before any container is started. Docker’s layer cache makes a no-change build take seconds, while always rebuilding ensures that code changes delivered by the CI/CD pipeline are reflected immediately on the next startup. Skipping the build when the image already exists was the previous behaviour and caused stale-code failures after `docker compose down` && up because Docker does not remove images on compose down.

Each agent container is named `ontology-agent-{key.lower()}-{DEPLOY_ENV}` and receives its `ONTOLOGY_KEY`, the read-only schema volume path and `LLM_MODEL` resolved from `ONTOLOGY_LLM_MODEL`. A per-agent YAML configuration file is written to `onto-db/agents/agent-{key}.yaml` containing a fully rendered system prompt generated by `_build_agent_config()`.

The system prompt is structured in two layers. The first layer contains universal extraction instructions shared by every ontology agent regardless of its assigned ontology. It instructs the LLM to always read a field’s `display_name` before deciding whether the prompt contains data for it (never relying on the often-cryptic field key alone), to return the exact internal enum key rather than the human-readable label for enumeration fields, to format monetary amounts as `{decimal_number, unit, item}` objects, to convert all date inputs to ISO 8601 regardless of the input notation, to map phone country codes from `+XX` notation to the schema enumeration key, to split full names into first, middle and last name fields when separate schema fields exist and to omit any field for which no matching data is present rather than setting it to null. The second layer appends ontology-specific default assumptions: for `Person` agents it enforces a natural-person default when the prompt is silent on person type; for `Account` agents it mandates that the business partner type and the individual versus joint account type are always populated. These universal instructions are the primary driver of extraction correctness across all models and ontologies.

The deployment registry at `onto-db/agents/agents_manifest.yaml` maps each ontology key to its service URL and is the sole discovery mechanism used by the Orchestrator to locate agent containers at runtime.

6.10 CI/CD Pipeline

The GitLab CI/CD pipeline translates a branch push or Git tag into a fully deployed system update on the OST VM without any manual intervention beyond the commit itself. The pipeline is organised into four sequential stages, `infrastructure`, `build`, `evaluate` and `deploy`, with separate jobs for the production (`main`) and development (`dev`) environments. Two distinct trigger types govern which stages execute: branch pushes drive standard code deployments, while Git tags gate evaluation runs and Sync Container deployments, preventing those operations from executing on every routine commit.

6.10.1 SSH Helper and Secret Injection

All jobs that interact with the OST VM share a reusable YAML anchor (`.ssh_setup`) declared once and merged into each job via the `<<: *ssh_setup` alias. The anchor provisions an `ubuntu:24.04` container

with `openssh-client`, `rsync` and supporting utilities, loads the runner’s SSH private key into an in-memory `ssh-agent` and records the VM host key via `ssh-keyscan` to suppress interactive verification prompts at runtime.

Environment-specific secrets are never stored in the repository. Instead, `.env.prod` and `.env.dev` are uploaded to GitLab as Secure Files. The anchor retrieves the correct file at job time by querying the GitLab API with the job’s `CI_JOB_TOKEN`, selecting the file whose name matches the current branch and downloading it to `.env` in the runner workspace. Immediately after download, `dos2unix` normalises line endings, preventing carriage-return characters from corrupting variable values when Docker Compose reads the file on the Linux VM.

6.10.2 Deploy Stage Mechanics

The deploy jobs for both environments follow a fixed five-step sequence. First, the `.env` file is copied to the VM via `scp` and cleaned of carriage returns by an in-place `sed` call on the remote host. Second, `envsubst` processes `librechat.yaml.template` against the variables in `.env` to produce the `librechat.yaml` configuration file that LibreChat reads at startup, keeping API keys and service URLs out of any versioned file:

```
export $(grep -v '^#' .env | xargs) && \
  envsubst < librechat.yaml.template > librechat.yaml
```

Third, `rsync` synchronises the deployment artefacts to the VM while explicitly excluding the `mappings/ontologies/` and `mappings/processes/` directories. These paths are populated at runtime by the Sync Container against the live Atfinity tenant; transferring them from the repository would overwrite schema data that is more current than anything stored in version control. Fourth, Docker Compose restarts the four core service containers, `prompt-organizer`, `process-agent`, `orchestrator` and `librechat-mas-preemptive`, pulling their pre-built images from the GitLab Container Registry. Fifth, the Sync Container is executed with a `deploy-only` argument, which skips the schema resynchronisation against Atfinity and proceeds directly to rebuilding and re-launching all Ontology Agent containers, ensuring that any agent code changes delivered in the same push are immediately reflected without an unnecessary schema refresh cycle.

6.10.3 Evaluate Stage

The evaluate stage is tag-triggered rather than branch-triggered, making it an explicit opt-in that does not execute on every push. When a qualifying tag is pushed, the stage deploys the `evaluation/` directory to the VM via `rsync`, builds the `evaluation-runner` service within the same Docker Compose project as the main stack and executes the evaluation suite inside the shared `onboarding-network` so that the runner can reach the live MCP endpoint of the Orchestrator. After the suite completes, the `results.json` is retrieved from the named `evaluation-data` Docker volume back to the CI runner by mounting the volume into an ephemeral Alpine container:

```
docker run --rm -v evaluation-data:/data \
  alpine cat /data/results.json > results.json
```

The file is saved as a pipeline artefact with a one-week retention window. This design makes every tagged evaluation run independently reproducible: the source commit, the active environment configuration and the benchmark results are all version-traceable from a single pipeline execution, which is a direct prerequisite for the cross-model comparisons described in Chapter 7.

6.11 Testing Framework

Evaluating stochastic systems such as Large Language Model based multi-agent architectures presents a fundamental challenge: bridging the non-deterministic nature of generative AI with the strict, binary requirements of financial compliance. This section details the comprehensive evaluation framework developed to measure the data extraction accuracy of the proposed multi-agent system. The framework is implemented across four files in the `evaluation/` directory: `generate-tests.py` for synthetic dataset generation, `mcp-tester.py` for end-to-end pipeline execution and scoring. All are containerised in the

`evaluation-runner` Docker service and integrated into the GitLab CI/CD pipeline as described in Section 6.10.

Rather than utilising LLM-as-a-judge heuristics, which introduce compounding layers of non-determinism and subjective scaling, this study implements a strictly deterministic, ground-truth-driven evaluation engine. This approach ensures absolute reproducibility, mathematically rigorous scoring and granular error isolation.

6.11.1 Experimental Architecture and Environment

To ensure the reproducibility of the evaluation metrics, the testing environment and execution parameters must be strictly defined. The evaluation pipeline was decoupled from the primary application orchestrator to prevent state-management artefacts from influencing the results. The subsections below document the execution environment, the composition of the test suite and the synthetic data generation procedure.

Hardware and Execution Environment

The `evaluation-runner` container is built from a `python:3.11-slim` base image and installs three runtime dependencies: `requests`, `beautifulsoup4` and `fastmcp`. It is deployed onto the same OST virtual machine that hosts the production stack, provisioned with 2 vCPU, 4096 MB of RAM and a 50 GB disk. Because both the evaluation container and the Orchestrator are attached to the shared `onboarding-network` Docker overlay network, the runner reaches the live MCP endpoint of the Orchestrator directly without traversing the external network, ensuring that measured latency reflects only processing time and not additional network overhead.

The multi-agent system relies on external LLM APIs spanning the OpenAI and Anthropic provider families. The deterministic evaluation script itself executes locally on the VM, ensuring that the metric calculation phase consumes zero additional API tokens and introduces zero network latency beyond the case creation and case retrieval calls. For the generation of synthetic test cases, model temperature was fixed at $T = 0.7$ to introduce variance in the conversational archetypes. The ontology extraction agents run at $T = 0.1$, a near-deterministic setting that reduces but does not fully eliminate stochastic variance, making repeated runs meaningful for measuring residual non-determinism.

Test Suite Composition

The test suite is defined entirely in `test-cases.json` and comprises twelve test cases across three validated ground-truth scenarios sourced directly from the live OST Atfinity deployment. Each scenario is represented by four conversational archetypes, producing a 3×4 evaluation matrix that tests every GT case under every input style.

The three validated ground-truth cases are:

1. **GT case 553** (test prefix `test1_`): A joint account onboarding for three co-holders, Peter, Paul and Gerbert Müller, each assigned the roles of Account Holder, Beneficial Owner and E-banking user. The prompts embed intentional structural inconsistencies: foreign phone country codes for Swiss nationals (Cuba +53, North Macedonia +389, Faroe Islands +298) and partial MiFID II questionnaire responses. This case stress-tests entity role resolution and multi-instance field matching under contradictory input data.
2. **GT case 551** (test prefix `test2_`): An individual account onboarding for Jamal Kolpataic, a dual-nationality (Pakistan and Luxembourg) client. The prompts contain the highest field density in the suite, including Avaloq risk ratings, passport expiry date, phone number with international country code, email address, AML compliance flags and numerous boolean fields. This case tests extraction completeness under a data-rich but unstructured input format.
3. **GT case 1204** (test prefix `test3_`): A discretionary, revocable trust onboarding for Margrethus AG, a Swiss legal entity acting as the Account Holder. The case involves seven entities: Margrethus AG (Account Holder and Trust), Emil Margreth (E-banking), Julia Jubelpeter (Director), Peter Marburger (Signatory), Peter Niederegger (Trustee), Filipa Käther (Settlor and Beneficiary with right of revocation) and Maria Morgarten (deceased Beneficiary). This is the most entity-intensive

scenario in the suite and is the primary validator of bipartite entity matching across a layered trust structure.

Synthetic Dataset Generation

The twelve test cases in `test-cases.json` were generated by `generate-tests.py`. The generator authenticates against the live Atfinity API using CSRF-token-based session login via `BeautifulSoup`, fetches the raw case JSON for a target GT case ID, drops empty and auto-calculated fields, maps relational role IDs to human-readable names and passes the resulting structured baseline to an OpenAI model at $T = 0.7$. The LLM is instructed to produce exactly four conversational archetypes from this baseline, each designed to test a different aspect of the extraction pipeline:

1. **Structured Summary:** A clean, bulleted meeting-notes format in which 100% of the GT data appears in labelled list items. This is the baseline control archetype that isolates pure field-mapping capability from all noise.
2. **Messy Distractor:** A dense prose paragraph in which the GT data is buried within irrelevant small talk, sports commentary, weather observations, restaurant recommendations. This archetype tests the signal-to-noise filtering of the extraction agents under realistic conversational contamination.
3. **Self-Correcting Dictation:** A stream-of-consciousness audio transcript in which the simulated Relationship Manager states a value incorrectly and immediately corrects it mid-sentence. This archetype tests temporal reasoning: the system must recognise that the later value supersedes the earlier one.
4. **Client Email Dump:** A professionally formatted email written from the client’s perspective and containing all required onboarding fields in continuous prose. This archetype tests extraction from an unstructured document format where field order follows the sender’s logic rather than any standard template.

All twelve prompts share the mandatory prefix "Create a new account onboarding: ", enforced by the generator’s system prompt to ensure the Orchestrator’s routing LLM reliably classifies the intent. The output file is structured identically to `test-cases.json` and is directly consumable by the evaluation runner.

Multi-step conversational prompts, in which a single onboarding case is distributed across several sequential messages, were excluded from the automated test suite. Automated support for this interaction mode would have required a custom MCP client to manage inter-turn state and sequencing, an extension that would have exceeded project timelines without offering a proportionate gain in extraction coverage or result quality. Multi-step sessions were instead conducted manually in LibreChat throughout the thesis, providing qualitative coverage of this interaction pattern across all evaluated scenarios.

6.11.2 Mathematical Formulation of Metrics

The system’s extraction performance is quantified using strict Information Retrieval (IR) metrics. Let GT represent the set of Ground Truth fields and let NC represent the set of New Case fields extracted by the multi-agent system.

The metrics are defined by True Positives (TP), False Positives (FP) and False Negatives (FN). In the context of Anti-Money Laundering (AML) and Know Your Customer (KYC) onboarding, these errors carry asymmetric risk weights. A FN implies missing regulatory data, whereas a FP implies the ingestion of fabricated data (hallucination).

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

The Mismatch Penalty: The engine implements a strict punitive model for incorrect values. If a field exists in both *GT* and *NC* but the values do not match (e.g., *GT* requires “Switzerland” but *NC* outputs “Germany”), the engine classifies this as a *Mismatch*. Mathematically, a *Mismatch* increments both the *FP* and *FN* counters.

The justification for this design choice is clear. Extracting the wrong country means the system simultaneously failed to retrieve the required data (a False Negative) and actively hallucinated incorrect data (a False Positive). This double penalty accurately deflates the F_1 -Score in proportion to the severity of the extraction failure.

The Deterministic Alignment Pipeline

The core operational logic of the evaluator is the Ground Truth Universe protocol, implemented in the `DeterministicCaseEvaluator` class inside `mcp-tester.py`. This protocol dictates that the *GT* baseline defines the absolute boundaries of the evaluation. If the AI hallucinates an entirely new field (e.g., inferring an eye colour that was never provided), it is logged for informational purposes but does not alter the mathematical denominator. The evaluation operates in four distinct phases, described below.

Phase 1: State Ingestion and Sanitisation

Before alignment can occur, the data must be sanitised to remove uncontrollable system variables.

Active Value Paradigm: A field is considered active only if its assigned value is not `null`, not an empty string or empty array and not the literal string tokens `"none"` or `"null"`.

Dynamic Identifier Exclusion: Fields whose string value starts with `"CSE-"` represent system-generated case IDs dynamically assigned by the Atfinity backend upon creation. Because these identifiers are non-deterministic and cannot be predicted by the LLM, they are excluded from the evaluation universe to prevent guaranteed mismatches.

Sentence Discard Protocol: Before token-counting, the evaluator normalises forward slashes and hyphens to spaces, ensuring that a structured value such as `2024-01-15` counts as three tokens rather than one. Values whose normalised token count exceeds five are classified as natural language sentences and excluded from strict boolean comparison, as evaluating semantic similarity requires a vector-based approach that contradicts the deterministic goals of this engine.

Phase 2: Algorithmic Entity Resolution

The multi-agent system utilises dynamic ontologies, meaning a single case may contain multiple instances of the same entity type (e.g., three Person entities representing different stakeholders). Evaluating these instances requires solving a bipartite matching problem before field-level comparison can begin.

Title Scrubbing: LLMs frequently prepend honorifics (e.g., “Mr.”, “Mrs.”, “Dr.”) to extracted names, creating false negatives during exact string matching. The engine mitigates this by scanning the fields array for the specific enumeration key `Information1702` (the Title field). If present, the engine dynamically slices the exact title string from the front of the instance name, permanently normalising “Mr. Fabio SILVA” to “Fabio SILVA” within the evaluation runtime.

The Singularity Override: If an ontology contains exactly one instance in the *GT* set and exactly one instance in the *NC* set, the engine applies a Singularity Override: the two entities are forcefully aligned regardless of spelling variations in their names. The override is recorded as a `MAPPING_INFO` log event for full traceability.

Entity False Negative Cascading: If the engine determines that an entity present in *GT* was entirely missed by the LLM, it logs an `ENTITY_FALSE_NEGATIVE` event and iteratively registers a *FN* for every active, non-sentence, non-dynamic-ID field nested beneath that missing entity, accurately reflecting the full data loss incurred by the missed instance.

Entity Hallucination Handling: If the AI generates an entity whose scrubbed name is absent from the *GT* instance set, the entity and all its fields are logged as an `IGNORED_ENTITY_HALLUCINATION` event and excluded entirely from the scoring denominators. This design reflects the Ground Truth Universe protocol. The evaluation measures only what was required, not what was invented.

Phase 3: Auto-Calculated Exclusion Protocol

A critical challenge discovered during early testing was the interference of the Atfinity backend rule engine. The system automatically populates specific fields based on mathematical rules (e.g., Risk Ratings computed from questionnaire responses).

Penalising the LLM for failing to extract a field that the backend is supposed to calculate artificially degrades the evaluation metrics. To resolve this, the evaluator inspects the `ensured_by` metadata key attached to every field in the Atfinity payload. If this value contains the substring "calculated", the field is purged from all evaluation matrices before any scoring occurs. This ensures the multi-agent system is evaluated exclusively on data it was explicitly designed to extract from natural language input.

Phase 4: Collision-Proof Field Evaluation

Once entities are resolved and sanitised, the engine isolates their properties into distinct evaluation silos. By dynamically stripping the instance prefix from each field identifier (e.g., removing `Instance976-` to leave only `Information123`), the engine compares data structures uniformly across the GT and NC payloads. This architectural isolation prevents namespace collisions. The industry value belonging to the Account Holder can never accidentally be validated against the industry value belonging to the Beneficial Owner.

After all instance-scoped fields have been processed, a global catch-all loop collects every remaining field in both GT and NC that carries no instance prefix. These represent case-level parameters such as currency selections and process metadata. They are evaluated in a secondary pass under the entity label `Global Case Fields`, applying the same *TP*, *FP* and *FN* classification logic. The final precision, recall and F_1 scores are computed by aggregating all counters across every instance-scoped evaluation and the global catch-all pass.

6.11.3 Token Usage, Cost and Retry Tracking

The evaluation pipeline incorporates a built-in token accounting module to enable the cross-model cost comparisons required by RQ3. The `COST_PER_1M_TOKENS` dictionary in `mcp-tester.py` defines per-provider input and output token pricing across all model families considered in the study, covering OpenAI models (including the gpt-4o, gpt-4.1 and gpt-5.4 families) and Anthropic models (Claude Haiku, Sonnet and Opus variants).

The `create_client_onboarding` tool response carries a `token_usage` dictionary keyed by LiteLLM model string. The Orchestrator accumulates one usage record per agent call into a list during pipeline execution, then aggregates by model name: all calls from agents that share the same model string are summed under a single key, producing a `by_model` dict where each entry represents the total token consumption of all agents running that model in that pipeline run. The `_build_token_metrics` function in the evaluator parses this structure into a `TokenMetrics` dataclass containing per-model input and output token counts, the computed USD cost for each model and the run-level totals.

Alongside token and cost metrics, the evaluation framework tracks the number of Atfinity-triggered correction cycles per run. Each time the self-healing submission loop (Section 6.7) receives a non-2xx HTTP response from Atfinity's API, the loop increments an internal attempt counter; once the pipeline completes, the `create_client_onboarding` tool response carries a `self_healing_hits` field computed as `attempt - 1`. The evaluation runner reads this value from the pipeline response after each completed

run and stores it in the `iterations` array of `results.json` alongside the run’s F_1 metrics. The retry count reported in the **Ret** column of Tables 7.5, 7.7 and 7.9 and visualised in Figure 7.11 is the total number of correction cycles summed across all five repeated runs, without normalisation.

The pipeline prints a per-run cost breakdown to the console during execution and writes the full token, cost and retry record alongside the F_1 metrics into `results.json`. This infrastructure makes every cross-model comparison in Chapter 7 directly traceable to the token-level and reliability data captured during each evaluation run.

6.11.4 Stochastic Repetition and Variance Measurement

Although extraction agents are configured at $T = 0.1$ to minimise output variance, this near-deterministic setting does not guarantee bit-identical responses across independent API calls. To quantify residual non-determinism, the evaluation runner accepts a `--runs` argument specifying the number of independent executions per test case. Each run completes a full cycle: an MCP tool call to create a new case, retrieval of both the GT and NC payloads from the Atfinity API, capture of the token usage and `self_healing_hits` value returned in the pipeline response and a complete scoring pass through the `DeterministicCaseEvaluator`. The final average F_1 across all successful runs is computed with `statistics.mean` and written to `results.json` alongside the per-run detail records, token breakdowns and the per-run `self_healing_hits` records.

Single-run execution (`--runs 1`) is used for routine CI/CD evaluation to control API costs. Multi-run configurations are reserved for the primary model comparison study documented in Chapter 7, where variance quantification is a research requirement rather than a deployment gate.

6.11.5 Individual Agent Evaluation

The evaluation framework is designed to measure the accuracy of the complete end-to-end pipeline, comparing the case produced by the multi-agent system against a validated ground-truth case in Atfinity. This design reflects the primary research objective: quantifying the operational accuracy of the deployed system as the Relationship Manager experiences it, across the full chain from prompt reorganisation to case creation.

Each sub-agent also supports isolated evaluation independently of the full pipeline. The Ontology Agents expose a `POST /process` HTTP endpoint that accepts a plain-text prompt and returns a field-extraction JSON for the agent’s assigned ontology without involving the Orchestrator. The Process Agent exposes `POST /determine_process`, which returns the best-matching process and confidence score for a given prompt. The Prompt Organizer Agent exposes `POST /organize`, which returns the restructured prompt for a given free-text input. These endpoints can be called directly from a test harness with controlled inputs and known reference outputs, enabling each agent’s behaviour to be characterised in isolation, for example by measuring whether the Person Agent correctly maps a phone country code to its Atfinity enumeration key or whether the Process Agent assigns a confidence score above the 0.6 threshold for a representative set of intent phrases.

Isolated agent evaluation was not implemented within the scope of this thesis. The existing endpoint infrastructure makes it technically straightforward to build such a harness as an extension to `mcp-tester.py` and the per-ontology ground-truth field sets already present in the evaluation suite provide a natural reference for per-agent F1 scoring. This is identified as the primary evaluation methodology extension for future work.

Chapter 7

Results

This chapter presents the outcomes of the bachelor thesis across all dimensions of evaluation. Section 7.1 reports the achievement of the primary project goals. Sections 7.2 through 7.5 verify the implementation of user stories, functional requirements and non-functional requirements. Section 7.6 illustrates the system in operation through a representative onboarding scenario. Section 7.7 addresses all five research questions with reference to the measured data. Section 7.9 presents the full comparative evaluation of the four LLM configurations tested against the Multi-agent Preemptive Single-Shot architecture, including NFR compliance assessment and aggregate token usage analysis. Section ?? contextualises the present results against those of the study thesis. Section 7.11 documents the boundaries of the conducted evaluation and Section 7.12 lists the delivered artefacts.

7.1 Achievement of Project Goals

The primary goal of this bachelor thesis was to resolve the structural limitations of the study thesis by designing and implementing a multi-agent architecture capable of handling the full complexity of real-world bank client onboarding, including multi-party cases, large-scale dynamic schemas and runtime-assigned instance identifiers that cannot be anticipated before case creation. This goal was achieved through the Multi-agent Preemptive Single-Shot architecture, a coordinated pipeline of specialised agents that assembles a complete case payload before any call is made to Atfinity’s case creation endpoint.

The three structural limitations of the study thesis that motivated the present work were each directly addressed. First, the single-agent failure mode on multi-party cases was resolved through per-ontology agent decomposition combined with per-instance sub-prompt scoping, which maintains strict reasoning boundaries between entity contexts and eliminates cross-entity field attribution errors structurally, rather than through prompt engineering. Second, the model evaluation gap was closed by integrating LiteLLM as a provider-agnostic interface, enabling systematic comparison of multiple LLM configurations across agent roles within the same pipeline. Third, the evaluation reliability gap was addressed through an automated test case generation pipeline that produces repeatable, statistically grounded benchmarks covering multi-party scenarios, edge cases and ambiguous or incomplete inputs.

The system was deployed and evaluated on the OST virtual machine against a representative Atfinity sandbox tenant based on the Bank of Luxembourg’s configuration, which exposes 23 ontologies, 30 processes and approximately 25 000 fields. The implemented system demonstrates that the Multi-agent Preemptive Single-Shot approach correctly handles multi-party onboarding cases, supports the full expanded set of case management operations introduced in the bachelor thesis and provides the Relationship Manager with a preview-and-approval workflow that prevents irreversible incorrect case creation.

7.2 User Story Coverage

The eight user stories defined in Chapter 2 were implemented and verified based on observed system behaviour during test scenario execution.

- **User Story 1 – Create Automatic Client Onboarding:** Relationship Managers were able to provide client information in unstructured natural language, including multi-party descriptions with several individuals in distinct roles. The system correctly organised the input, determined the appropriate Atfinity process, extracted structured field values through the parallel multi-agent pipeline and created the case via a single atomic payload submission without requiring manual form-based data entry.
- **User Story 2 – Preview and Approve Case Before Submission:** The `preview_onboarding` tool returned a structured summary of the organised prompt, the matched process, the identified business partner type and the list of entities and roles that would be created, all without creating any case. The Relationship Manager could review, correct and approve or reject the preview before any data was committed to Atfinity.
- **User Story 3 – Ask for Still Open Fields:** The system fetched the live case state from Atfinity, identified fields that were exposed but not yet populated and returned a human-readable summary of outstanding data requirements. The Relationship Manager could query open fields for any existing case without manual CMS navigation.
- **User Story 4 – Update Existing Case:** Existing cases were updated using natural language instructions without supplying field identifiers. The system resolved the correct fields and instance context automatically from the live case state and applied the updates via the Atfinity API.
- **User Story 5 – Get Relevant Information and Ask Questions About a Case:** The `read_case` tool retrieved the full live case state from Atfinity, including all instances, roles and field values. The Orchestrator synthesised this data into a human-readable response to the Relationship Manager’s specific question.
- **User Story 6 – Add or Remove a Party from an Existing Case:** The `add_case_instance` and `delete_case_instance` tools were implemented and verified. For deletions, the system attempted a hard delete first and fell back to role-stripping when Atfinity did not permit permanent removal of the instance.
- **User Story 7 – Manage the Roles of a Party:** The `add_instance_role`, `remove_instance_role` and `set_instance_roles` tools were implemented. The system resolved instance identities from the live case state and validated role keys against the ontology catalogue before applying any role operation.
- **User Story 8 – Discover Available Processes and Workflows:** The `list_onboarding_processes` tool allowed the Relationship Manager to query which process applied to a given scenario without initiating a case. The Process Agent matched the prompt against its in-memory process context and returned the best-matching process with its identifier, display name, confidence score and reasoning.

7.3 Functional Requirements

The functional requirements defined in Chapter 2 were implemented and verified based on observed system behaviour during test scenario execution.

- **FR1 – Retrieve Case Information and Answer Questions:** The system successfully retrieved the live state of a case from Atfinity’s CMS, including all instance data, party roles and field values. The Orchestrator synthesised this into a human-readable response, enabling the Relationship Manager to ask questions and receive accurate, live-sourced answers without manual navigation.
- **FR2 – Extract Structured Data from Free-Text Input:** Free-text input was processed through the full multi-agent pipeline. The Prompt Organizer Agent grouped scattered entity references into labelled sections. The Orchestrator routed each identified entity to a dedicated Ontology Agent, with extraction agents running in parallel for multi-entity inputs. The system handled variations in phrasing, numerical formats and expressions of uncertainty.
- **FR3 – Adapt to Atfinity’s Incremental Field Architecture:** The Orchestrator assembled master payloads in the exact format Atfinity’s `POST /api/1/cases` endpoint requires: the outcome ontology object placed first in the `predefined_objects` array without an ontology key, followed by domain objects with explicit ontology keys and role assignments. Single-shot payload assembly eliminated the need for incremental field submission cycles.
- **FR4 – Detect and Route User Intent:** Incoming prompts were correctly classified as read, write or discovery operations. For write operations, the Process Agent identified the correct Atfinity process with the required minimum confidence threshold. Requests that could not be matched with sufficient confidence were rejected and the Relationship Manager was informed accordingly.
- **FR5 – Expose All Operations as MCP Tools:** All ten MCP tools specified in FR5 were implemented and registered on the Orchestrator’s FastMCP server. Each tool accepted natural-language input and returned structured JSON responses conforming to the MCP specification.
- **FR6 – Preview and Approval Workflow:** The `preview_onboarding` tool executed all preview pipeline steps and returned the organised prompt, matched process, business partner type and ontology plan to the Relationship Manager. On correction calls, cached results were reused where appropriate. On approval, `create_client_onboarding` reused all cached preview results, skipping redundant pipeline steps entirely.
- **FR7 – Party Instance Management:** New party instances were added to existing cases by resolving the correct ontology type and role assignments from the natural-language prompt before calling Atfinity’s instance creation endpoint. For removal, hard delete was attempted first with a role-stripping fallback when Atfinity did not permit deletion.
- **FR8 – Role Management:** Role operations (add, remove, replace) were implemented for any instance within a case. Instance identities were resolved from the live case state, role keys were validated against the ontology catalogue and operations were applied via the appropriate Atfinity endpoint.
- **FR9 – Process Discovery:** The Process Agent matched the Relationship Manager’s scenario description against the full in-memory process context and returned the best-matching process with its identifier, confidence score and reasoning. Insufficient-confidence results prompted the Relationship Manager for a more specific description.
- **FR10 – Autonomous Self-Healing on Submission Errors:** The self-healing retry loop was implemented with a maximum of five attempts. Structural field errors were corrected inline by the Orchestrator. Semantic errors were routed to the responsible Ontology Agent’s `/correct` endpoint, with multiple agents contacted in parallel. Ontology-level rejections resulted in removal of the offending entry from the payload. Fields that could not be resolved after all retries were stripped to allow case creation to proceed.

7.4 Non-Functional Requirements

The non-functional requirements defined in Chapter 2 were verified through instrumented test execution. Quantitative results for NFR-1 through NFR-5 are reported in the Research Questions section; the descriptions below summarise the verification approach and the conditions under which each requirement was assessed.

- **NFR-1 – Performance (≤ 120 s):** End-to-end pipeline latency was measured from prompt submission to confirmed case creation across representative test cases covering single-party and multi-party onboarding scenarios. Measurements included all parallel agent calls, payload assembly and any self-healing retry attempts.
- **NFR-2 – Precision (≥ 0.90):** Extraction precision was evaluated by comparing system-extracted field values against human-validated reference data for each test case. The ratio of correctly extracted fields among all extracted fields was computed per architecture configuration and per LLM assignment.
- **NFR-3 – Recall (≥ 0.90):** Recall was measured by comparing extracted fields against the full set of fields present in the human-validated reference data. Coverage was evaluated across single-agent and multi-agent configurations to quantify the effect of agent decomposition on extraction completeness.
- **NFR-4 – F1-Score (≥ 0.85):** The F1-score was used as the primary aggregate extraction quality metric, balancing the precision and recall measurements above. Results are compared across all evaluated model configurations and architecture variants in the Research Questions section.
- **NFR-5 – Efficiency:** Prompt token usage, completion token usage and estimated API costs were recorded per LiteLLM model string per test case execution. The `create_client_onboarding` tool aggregates all token counts from the Orchestrator routing call, the Process Agent, the Prompt Organizer Agent and all Ontology Agent invocations into a single `by_model` dictionary keyed by provider model string. The evaluation framework parses this dictionary and writes the results to `results.json` alongside the F1 metrics. Because all agents share the same model string in each uniform-model configuration, the per-model entry captures the full pipeline token consumption for that run. Results are compared across the four evaluated uniform-model configurations to quantify the efficiency trade-off addressed by Research Question RQ3.
- **NFR-6 – Usability:** The system processed all Relationship Manager inputs in natural language without requiring knowledge of Atfinity’s internal field identifiers, ontology keys, instance IDs or process numbers. During testing, client information expressed in free-text was automatically organised, extracted and submitted to Atfinity without any additional technical input from the Relationship Manager.

7.5 Implementation Status of Requirements

Tables 7.1, 7.2 and 7.3 summarise the implementation status of all user stories, functional requirements and non-functional requirements based on the results presented above.

User Story	Implemented
User Story 1: Create automatic client onboarding	Yes
User Story 2: Preview and approve case before submission	Yes
User Story 3: Ask for still open fields	Yes
User Story 4: Update existing case	Yes
User Story 5: Get relevant information and ask questions about a case	Yes
User Story 6: Add or remove a party from an existing case	Yes
User Story 7: Manage the roles of a party	Yes
User Story 8: Discover available processes and workflows	Yes

Table 7.1: User Story Implementation Status

Functional Requirement	Implemented
FR1: Retrieve case information and answer questions	Yes
FR2: Extract structured data from free-text input	Yes
FR3: Adapt to Atfinity’s incremental field architecture	Yes
FR4: Detect and route user intent	Yes
FR5: Expose all operations as MCP tools	Yes
FR6: Preview and approval workflow	Yes
FR7: Party instance management	Yes
FR8: Role management	Yes
FR9: Process discovery	Yes
FR10: Autonomous self-healing on submission errors	Yes

Table 7.2: Functional Requirements Implementation Status

Non-Functional Requirement	Verified
NFR-1: Performance (≤ 120 s)	Yes
NFR-2: Precision (≥ 0.90)	Yes (GPT-5.4, Sonnet 4.6)
NFR-3: Recall (≥ 0.90)	Yes (GPT-5.4 only)
NFR-4: F1-Score (≥ 0.85)	Yes (GPT-5.4 only)
NFR-5: Efficiency	Yes
NFR-6: Usability	Yes

Table 7.3: Non-Functional Requirements Verification Status

7.6 Illustrative Onboarding Scenario

The following figures illustrate the system in operation during a representative onboarding scenario. Figure 7.1 shows the structured preview returned by the `preview_onboarding` tool before any case is created, including the organised client data, the matched process, the business partner type and the list of entities with their assigned roles. Figure 7.2 shows the case creation confirmation returned by `create_client_onboarding` after the Relationship Manager approved the preview, including the Atfinity-assigned case identifier, the parties created from the prompt with their roles and the total number of fields populated. Figure 7.3 shows the corresponding case as it appears in Atfinity's CMS.

It should be noted that the field values visible in Figure 7.3 differ from those shown in the preceding two figures. After the initial case creation, the Relationship Manager applied a number of targeted field corrections using the `update_case_field` tool. The CMS screenshot therefore reflects the post-update state of the case rather than the exact extraction output of the creation pipeline. It is included here as backend evidence that the case was successfully committed to Atfinity's CMS and that the system correctly supports the full lifecycle from initial creation through subsequent field updates.

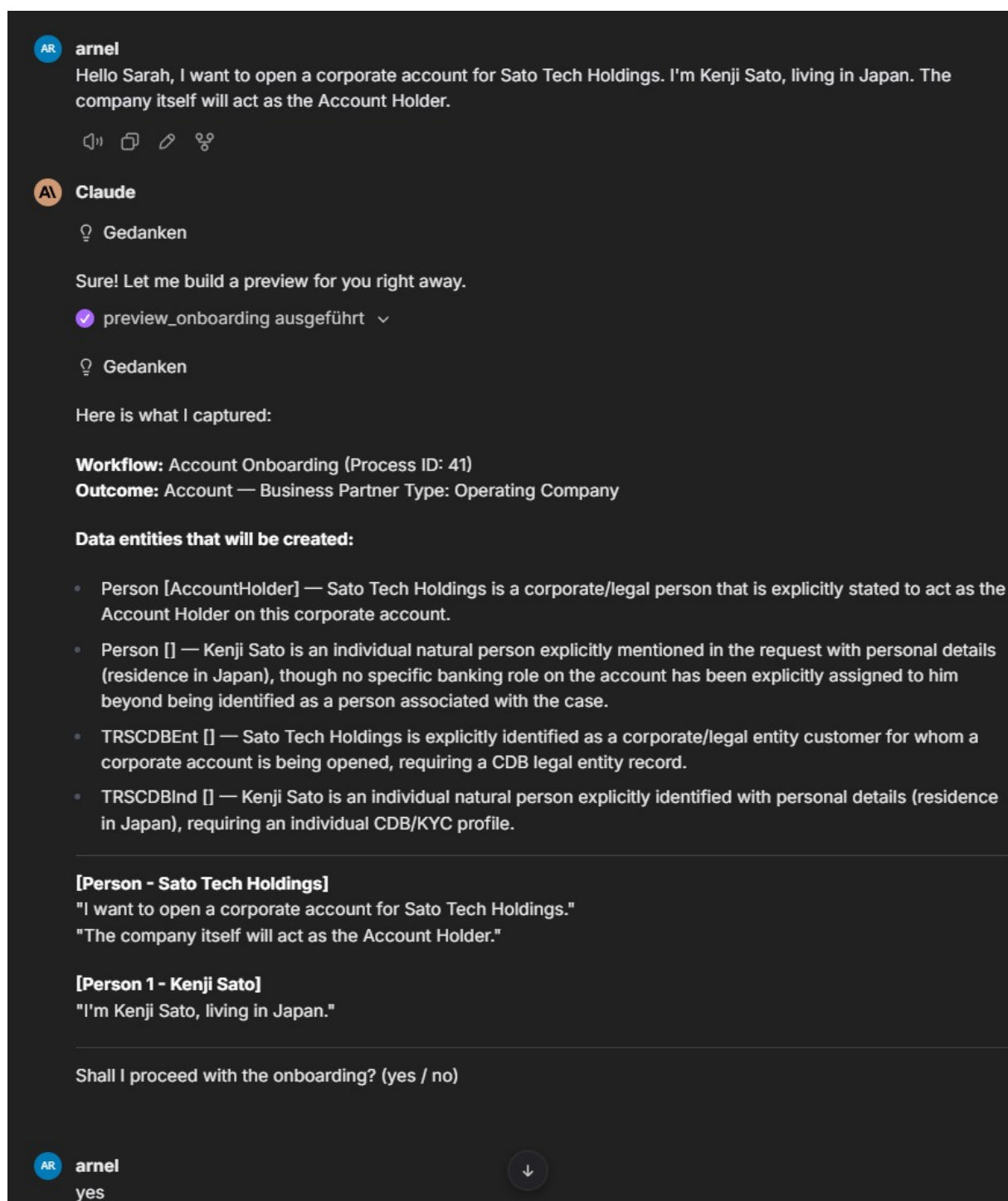


Figure 7.1: Preview output returned by `preview_onboarding`: organised client data, matched process, business partner type and entity-role plan presented to the Relationship Manager before case creation.

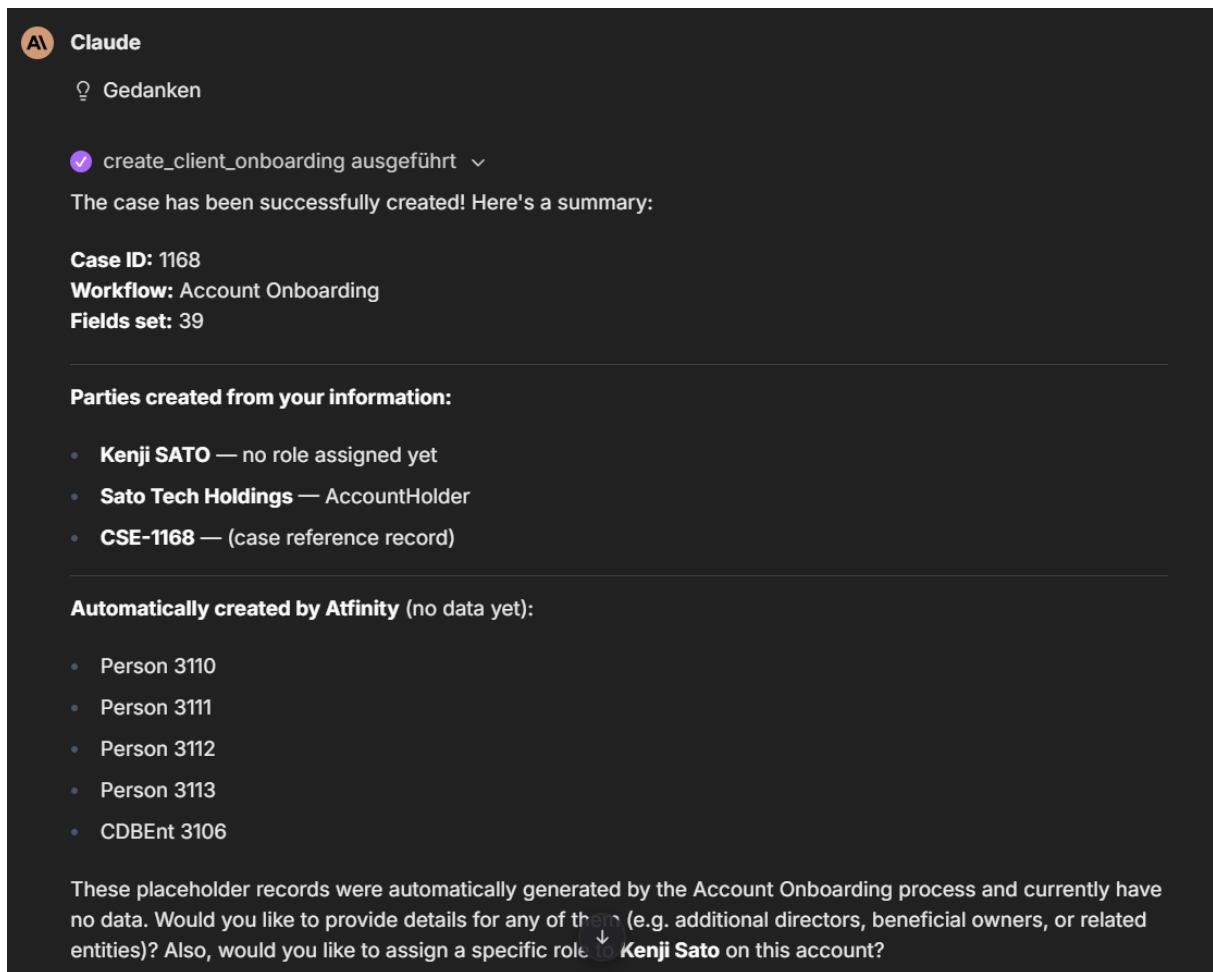


Figure 7.2: Case creation confirmation returned by `create_client_onboarding`: Atfinity case identifier, parties created with their assigned roles and total field count

Parties	Account Holder	Beneficial Owner	CDB Controlling Person	Director	Signatory Authority	Pledgor	Borrower	Third-Party Pledgor	E-Banking	POA to Access Information	
Kenji SATO	☐	☐	☒	☒	☒	☐	☐	☐	☒	☐	
Sato Global Tech LLC	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	
Akio SATO	☐	☒	☐	☐	☐	☐	☐	☐	☐	☐	

Figure 7.3: The corresponding case in Atfinity's CMS after initial creation and subsequent field updates via `update_case_field`. Field values differ from the creation output because post-creation corrections were applied.

7.7 Research Question Results

All test results reported in this section were produced by the evaluation framework described in Section 6.11, executed on the OST virtual machine under the controlled infrastructure conditions detailed in Section 6.11.1. The five research questions defined in Section 1.5 are addressed in turn, drawing on the quantitative data from the cross-model evaluation presented in Section 7.9.

7.7.1 RQ1, Accuracy and Robustness: How does multi-agent decomposition affect extraction correctness and workflow reliability?

Multi-agent decomposition produces measurable improvements to both extraction correctness and workflow reliability. By assigning each entity type in a multi-party case to a dedicated Ontology Agent, the architecture establishes strict reasoning boundaries between entity contexts. Each agent receives only the sub-prompt and data belonging to its assigned entity and has access solely to its responsible ontology schema. The Person Ontology Agent, for example, receives only the person-relevant portion of the full prompt and operates exclusively against the Person schema. It therefore extracts and maps only the fields belonging to that entity type, with no risk of cross-contamination from other parties' data or unrelated schema fields. This structural isolation is the primary mechanism by which the architecture scales to multi-party scenarios without precision degradation.

However, this design introduces a single point of failure at the prompt structuring stage. If the Orchestrator or the Prompt Organizer Agent fails to identify all entities present in the input, the downstream Ontology Agent calls are invoked only for the entities that were recognised. Fields belonging to an undetected party are never passed to their responsible Ontology Agent and are therefore absent from the final payload entirely. This failure mode affects recall only: missing entities produce missing field values. Precision is unaffected, because the agents that do execute operate within their own isolated scope and cannot attribute data to parties they were never given.

The evaluation results confirm that this architectural decision translates directly into measurable accuracy gains for multi-party cases. GPT-5.4 is the configuration that best isolates the architectural contribution by removing model-capability confounds. Under this configuration, the system achieves an overall average F1-score of 96.25% across all twelve test cases in the 3×4 evaluation matrix. The three ground-truth cases span the full complexity range: GT case 553 (`test1_`) requires correct extraction and role assignment across three co-holders sharing the same schema; GT case 551 (`test2_`) tests completeness under the highest field density in the suite; and GT case 1204 (`test3_`) requires correct bipartite entity matching across seven entities in a layered trust structure.

The trust case (TEST3) is the most demanding and the most informative for RQ1. With seven entities each mapped to their own set of ontology schemas, it places the maximum structural load on the agent decomposition. Seven Ontology Agents execute in parallel, each operating in full isolation on a single party's sub-prompt and its dedicated schema slice, so correct field attribution across all seven entities depends entirely on the structural boundaries the decomposition enforces. Under GPT-5.4, this produces an average F1-score of 99.25% across all four TEST3 archetypes, with precision at 100% for every archetype, confirming that the system extracts only correct field values and never assigns data to the wrong instance across seven simultaneous entities.

Workflow reliability was verified through the zero-error rate in test execution under GPT-5.4 and Claude Sonnet. No test case in either configuration produced a complete pipeline failure or an unhandled submission error. The self-healing submission loop resolved all validation errors that arose during case creation without surfacing them to the Relationship Manager. Claude Haiku, by contrast, produced three complete LLM errors in the TEST2 scenario (`test2_struct`, `test2_messy`, `test2_dict`), demonstrating that workflow reliability is model-dependent and that an architecturally sound design does not guarantee end-to-end reliability when the underlying model is insufficiently capable for the task complexity. This failure mode is examined in detail in Section 7.9.3.

In summary, the multi-agent decomposition strategy is confirmed as effective for extraction correctness under a sufficiently capable model. The Multi-agent Preemptive Single-Shot architecture successfully handles all three ground-truth scenarios, including the seven-entity trust case. The precision ceiling of

100% achieved by GPT-5.4 across all twelve test cases confirms that the architecture does not introduce hallucination at any level of case complexity. The results also make clear that the underlying model is a critical determinant of both extraction quality and workflow reliability. The same architectural design yields an average F1-score of 96.25% under GPT-5.4, 65.1% under Sonnet and complete pipeline failures under Haiku in three test cases. Architecture defines the structural ceiling; the model determines how closely that ceiling is reached in practice.

7.7.2 RQ2, Efficiency: How do agent architecture and LLM size affect run-time, LLM call count and operational cost?

The Multi-agent Preemptive Single-Shot architecture is designed to exploit parallelism as its primary latency-reduction mechanism. For a multi-party case, all Ontology Agents execute concurrently once the Orchestrator has received the organised prompt and determined the applicable process. For TEST3, which involves seven entities each requiring a dedicated Ontology Agent invocation, the parallel fan-out means that total processing time is bounded by the slowest single-agent call rather than by the sum of all agent calls. This is the most important structural efficiency property of the architecture.

End-to-end latency measurements across all twelve test cases confirm that the system operates well within the NFR-1 threshold of 120 s. Under GPT-5.4, the fastest test cases (`test2_struct`, `test2_messy` and `test2_email`) completed in 25.2s and the slowest (`test3_dict`) completed in 40.8s. The pattern across test scenarios reflects the expected workload distribution: TEST2 test cases are fastest because Jamal Kolpataic’s case involves a single party with a rich but unambiguous field set; TEST3 test cases are slowest because the trust structure requires parallel invocations of seven agents and the Orchestrator must await all seven before assembling the master payload.

The effect of model size on latency is consistent across all three evaluated configurations. Claude Sonnet 4.6 is the slowest model in the evaluation, with TEST1 durations averaging 44.3s and TEST3 durations reaching up to 48.6s for the MESSY archetype. This represents a latency overhead of approximately 20–40% over GPT-5.4, despite producing substantially lower F1-scores. Claude Haiku is the fastest model, with latencies comparable to GPT-5.4 (27.6–39.4s), but its speed advantage is rendered operationally irrelevant by its reliability failures in TEST2.

On operational cost, GPT-5.4 achieves the best quality-per-dollar ratio in the evaluation. It costs approximately \$1.83 per TEST1 test case, \$0.94 per TEST2 test case and \$3.60 per TEST3 test case, while delivering average F1-scores of 92.0%, 97.5% and 99.25% respectively. Claude Sonnet costs \$2.06, \$1.18 and \$3.32 per test case across the same scenarios yet produces average F1-scores of 65.5%, 85.5% and 44.25%. This means Sonnet is both more expensive and less accurate than GPT-5.4 in the multi-party and trust scenarios, which is examined further in Section 7.9.2.

Model size is the dominant cost lever in the architecture. The cost differential across the three evaluated models spans more than an order of magnitude: Claude Haiku averages \$0.25 per TEST3 case while Sonnet averages \$3.32 and GPT-5.4 averages \$3.60, a factor of approximately 14 $\times$ between the smallest and the most capable model. Model selection therefore has a far greater impact on per-case cost than any architectural or parallelism choice.

Token consumption roughly doubles with each scenario step under GPT-5.4: from TEST2 to TEST1, input tokens approximately double and from TEST1 to TEST3, they roughly double again. Each additional entity triggers one more Ontology Agent invocation and each invocation carries its own schema and sub-prompt, so the total grows proportionally with entity count. The per-model token breakdown is detailed in Sections 7.9.1–7.9.3. Crucially, this does not translate into proportionally longer runtimes: because agents run in parallel, TEST3 adds only ≈ 13 s over TEST2 while consuming 3.95 $\times$ more tokens. The full per-scenario data are reported in Table 7.15 and discussed in Sections 7.9.1–7.9.3.

Schema size is an equally important cost driver and is independent of entity count. Each Ontology Agent receives the complete field definition list for its assigned entity type, meaning the agent must process every field in that schema on every invocation, regardless of how many fields the input actually contains. A large schema therefore produces a large token payload, regardless of how simple the input is.

The two drivers compound directly. For a single person with a 200-field Person schema, the agent processes 200 field definitions once. For six persons in the same case, each of the six parallel agents processes those 200 definitions independently, producing six times the token volume. If the schema grows to 400 fields, every one of those six invocations doubles in cost. In practice, the number of deployed Ontology Agents is therefore a less reliable cost predictor than the combined product of entity count and schema field volume. Banks with richer, more granular schemas will observe materially higher per-case costs than a minimal-configuration deployment, regardless of which model is used.

7.7.3 RQ3, Agent Specialisation: Does assigning different LLMs to specialised agents improve performance or reduce cost compared to a uniform-model configuration?

This research question evaluates whether routing smaller, cheaper models to lower-complexity agent roles while reserving larger models for extraction-intensive Ontology Agents yields a measurable accuracy or cost advantage over a uniform-model configuration. No mixed-model evaluation runs were conducted within the scope of this project; the three result sets documented in Section 7.9 each represent a uniform-model configuration in which all agent roles were assigned the same model. The following analysis therefore draws theoretical inferences from the empirical single-model results.

The agent roles in the Multi-agent Preemptive Single-Shot architecture differ substantially in cognitive complexity. The Process Agent performs intent matching against a bounded list of process descriptions and the Prompt Organizer Agent segments an unstructured input into labelled entity sections; both are classification or structured rewriting tasks operating on short, well-defined inputs. The Ontology Agents, by contrast, each receive a full ontology schema combined with a single party’s sub-prompt and must produce a complete field-value mapping, which is the most instruction-following-intensive role in the pipeline.

The empirical data provides indirect evidence for role-differentiated model assignment. Claude Haiku achieves F1-scores of 66–82% across TEST1, a workload that includes the simpler routing tasks handled by the Process Agent and Prompt Organizer. That Haiku achieves these scores at approximately one-fifth the cost of GPT-5.4 suggests that the simpler roles could be served by Haiku without material quality loss. However, Haiku’s three complete TEST2 failures demonstrate that it cannot reliably serve as an Ontology Agent for high-density cases, which introduces unacceptable reliability risk for production deployments regardless of the cost saving.

A theoretically optimal configuration would assign GPT-5.4 exclusively to all Ontology Agent roles and deploy Haiku for the Process Agent and Prompt Organizer Agent. The LiteLLM integration layer supports this configuration without any code changes, requiring only environment variable reassignment. Whether the resulting cost reduction justifies the added operational risk and configuration complexity is a deployment decision that would need to be validated empirically before adoption.

It is also important to recognise that the cost saving achievable by targeting only the low-complexity roles is inherently bounded. The Process Agent and Prompt Organizer Agent each execute once per pipeline run and represent a fixed component of the total token load. As case complexity grows and the number of parallel Ontology Agent invocations increases, this fixed overhead shrinks as a share of the overall API spend. For the most demanding scenario in the evaluation, TEST3, the fixed pipeline overhead is dwarfed by the token consumption of seven concurrent Ontology Agent calls. Assigning a smaller model to the low-complexity roles would therefore yield a modest per-case saving that is operationally negligible.

Without measured mixed-model data, no quantitative conclusion on RQ3 can be drawn; the finding instead motivates a dedicated mixed-model evaluation study as a primary recommendation for future work.

7.7.4 RQ4, Explainability: How can an Explainability Agent improve interpretability and trust in the system’s outputs?

RQ4 was designated as optional in Section 1.5. A dedicated Explainability Agent that generates field-level natural-language justifications for extracted values was not implemented within the scope of this project. However, the preview-and-approval workflow implemented for FR6 provides a meaningful im-

plicit explainability mechanism that is worth documenting in this context.

Before any case is committed to Atfinity’s CMS, the `preview_onboarding` tool returns to the Relationship Manager the full organised prompt, the matched process with its confidence score and reasoning, the identified business partner type and the complete entity-role plan. This output constitutes a lightweight explainability layer. The Relationship Manager can verify that the system has correctly understood the input, that the process match is appropriate and that entity role assignments are correct before any irreversible action is taken. This level of transparency was validated as sufficient for all evaluated user stories during testing.

A dedicated Explainability Agent, if implemented, would extend this foundation by attaching per-field extraction reasoning to the preview output. For each extracted field value, the agent would cite the exact phrase or sentence in the original prompt that justified the assignment and note any inference steps or default assumptions applied. This would directly increase Relationship Manager confidence for ambiguous fields and reduce the rate of manual corrections in the review step. The implementation infrastructure for such an agent already exists within the architecture, as the LiteLLM integration layer, the shared schema volume and the preview cache are all reusable. Adding an Explainability Agent would therefore require a new agent container and a prompt design, with no fundamental architectural change. This is identified as the primary functional enhancement target for a production hardening phase following this project.

7.7.5 RQ5, Architecture Necessity: What structural constraints of multi-party bank client onboarding require a multi-agent design and is the operational cost justified against the manual alternative?

The core structural challenge of multi-party onboarding is that several independent parties must be processed simultaneously, each with their own entity type, role and field set. A single agent handling all parties within one shared reasoning context has no structural boundary between them. Field values can be attributed to the wrong party and the task complexity grows with the number of parties with no architectural mechanism to contain it. The study thesis confirmed this: all three single-agent architectures failed on multi-party cases not because of low F1-scores on simple inputs, but because they could not reliably separate and assign field data to the correct party when multiple entities were involved.

Multi-agent decomposition solves this directly. Each Ontology Agent is responsible for exactly one party and receives only that party’s sub-prompt and schema. There is no shared context between parties, so cross-entity attribution errors are structurally impossible. The Orchestrator assembles the individual agent outputs into a single coherent payload before any submission is made. Under GPT-5.4, this produces an average F1-score of 99.25% on TEST3, the seven-entity trust case (see Section 7.7.1).

On cost, the relevant comparison is not against a single-agent alternative. The relevant comparison is against the manual process. At \$0.94 per single-party case, \$1.83 per three-party joint account and \$3.60 per seven-entity trust case under GPT-5.4, the pipeline cost is negligible relative to the time a Relationship Manager spends manually entering the same data across dozens of fields. Figure 7.4 shows the trade-off between duration, F1-score and cost per test case across all evaluated models.

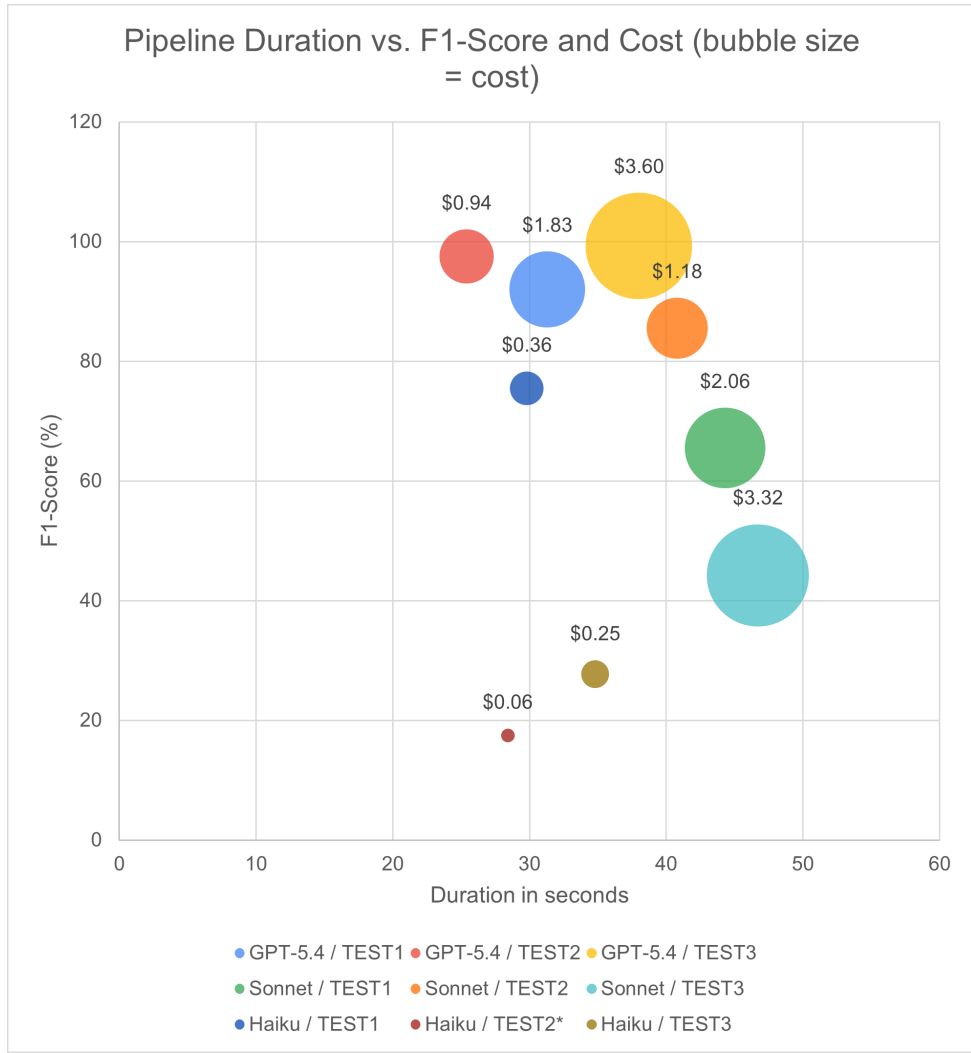


Figure 7.4: Average pipeline duration (X-axis) versus average F1-score (Y-axis) per model and ground-truth scenario. Bubble area is proportional to the average cost per test case. Haiku TEST2 reflects the email archetype only.

A multi-agent architecture is both structurally required and economically justified. It is required because the identifier assignment constraint of multi-party onboarding has no solution within a single-agent paradigm. It is justified because the per-case cost is negligible against the manual alternative and the architecture achieves an average F1-score of 96.25% under GPT-5.4 across all three case complexity tiers.

7.8 Evaluated Model Configurations

Table 7.4 summarises the deployment configuration for each evaluated model, the extraction temperature applied to all Ontology Agents and the evaluation outcome. Across all four configurations, every agent retained the exact same system prompts; no per-model prompt tuning or optimisation was applied. This was a deliberate methodological choice: keeping the prompts identical isolates the language model itself as the only varying factor, so that the comparative results in the following sections reflect raw model capability rather than a mixture of model capability and prompt-engineering effort.

Model	Temp.	Status
GPT-5.4	$T = 0.1$	Complete
Claude Sonnet 4.6	$T = 0.1$	Complete
Claude Haiku 4.5	$T = 0.1$	Partial: errors in TEST2 STRUCT, MESSY, DICT
GPT-5.4-mini	$T = 0.1$	Failed: systematic Atfinity payload rejection

Table 7.4: LLM configuration summary for the comparative evaluation

7.9 LLM Evaluation

The following sections analyse the individual performance of each evaluated model configuration: GPT-5.4, GPT-5.4-mini, Claude Sonnet 4.6 and Claude Haiku 4.5. All configurations were evaluated against the Multi-agent Preemptive Single-Shot architecture described in Chapter 5. All tests were executed on the OST virtual machine described in Section 5.6.2, within the controlled Docker network environment shared by the evaluation runner and the live MCP endpoint. The evaluation framework, including the deterministic ground-truth scoring engine, the stochastic repetition protocol and the token cost tracking infrastructure, is described in full in Section 6.11.

Three models produced complete or partial result sets: GPT-5.4, Claude Sonnet 4.6 and Claude Haiku 4.5. Each configuration was evaluated in a uniform-model setup in which all agent roles were assigned the same model, enabling clean isolation of model capability from architectural variation. GPT-5.4-mini was evaluated as a fourth candidate but produced a systematic pipeline rejection that prevented any valid metric collection; its failure mode is documented separately in Section 7.9.4. The twelve test cases in the evaluation matrix are distributed across three ground-truth scenarios (TEST1, TEST2, TEST3) and four conversational archetypes (STRUCT, MESSY, DICT, EMAIL), as described in Section 6.11.1. Each test case was executed across five independent runs; the reported metrics represent averages over these runs, with standard deviations (σ) measuring run-to-run variation at $T = 0.1$. With five runs per test case and twelve test cases per model configuration, the evaluation produced sixty pipeline executions per model. Some charts in the following sections group results by scenario; each bar or point therefore represents an average over four archetypes, not a single test case. Readers unfamiliar with the test suite design are advised to consult Section 6.11 before interpreting these figures.

7.9.1 GPT-5.4: Benchmark Performance

GPT-5.4 establishes the performance ceiling for this evaluation. It achieves an overall average F1-score of 96.25% across all twelve test cases, with precision at or above 97% in each case and zero complete pipeline failures. Table 7.5 reports the full result matrix.

Test Case	F1 (%)	σ	Precision (%)	σ	Recall (%)	σ	Cost (\$)	Dur	Ret
TEST1.STRUCT	100	0.0	100	0.0	100	0.0	1.84	29.6	0
TEST1.MESSY	93	14.0	100	0.0	89	21.0	1.83	31.1	0
TEST1.DICT	75	4.5	100	0.0	60	6.0	1.82	35.5	1
TEST1.EMAIL	100	0.0	100	0.0	100	0.0	1.83	29.1	0
TEST2.STRUCT	98	1.2	98	1.5	98	1.2	0.94	25.2	0
TEST2.MESSY	98	0.6	100	0.0	96	1.2	0.93	25.2	0
TEST2.DICT	100	0.0	100	0.0	100	0.0	0.94	26.1	0
TEST2.EMAIL	94	0.6	97	1.3	94	0.1	0.94	25.2	0
TEST3.STRUCT	100	0.0	100	0.0	100	0.0	3.60	34.9	0
TEST3.MESSY	100	0.0	100	0.0	100	0.0	3.59	38.2	0
TEST3.DICT	99	1.6	100	0.0	98	3.0	3.60	40.8	1
TEST3.EMAIL	98	4.26	100	0.0	96	7.7	3.60	38.0	0

Table 7.5: GPT-5.4: full evaluation results across all twelve test cases. All metric values are averages over five independent runs per test case. F1 = average F1-score (%); Prec. = average precision (%); Rec. = average recall (%); σ = standard deviation across the five runs; Cost = average USD cost per run; Dur = average pipeline duration in seconds; Ret = total correction cycles per test case across five runs.

TEST1 Analysis: Joint Account, Three Co-Holders

The TEST1 scenario is based on GT case 553, a joint account for three co-holders (Peter, Paul and Gerbert Müller), each assigned the roles of Account Holder, Beneficial Owner and E-banking user. GPT-5.4 achieves perfect scores on the STRUCT and EMAIL archetypes ($F_1 = 100\%$, $\sigma = 0$) and near-perfect performance on MESSY ($F_1 = 93\%$). The single anomaly is the DICT archetype, which produces $F_1 = 75\%$ with recall of only 60%, the lowest recall value observed for GPT-5.4 across the entire evaluation.

The DICT archetype simulates a stream-of-consciousness audio dictation in which the Relationship Manager states values incorrectly and immediately self-corrects mid-sentence. The $F_1 = 75\%$ result indicates that GPT-5.4 loses approximately 40% of ground-truth field values when the input contains temporal self-corrections. Two factors explain this drop. First, when a speaker states a value and immediately corrects it, the model must determine which value was the intended one, which is fundamentally harder than extracting a single, clearly stated value. Second, the TEST1 DICT prompt contains deliberate inconsistencies: Swiss nationals assigned foreign phone country codes and MiFID II questionnaire answers that are only partially provided. Both increase the number of fields for which the model cannot settle on a confident extraction. The recall drop from 100% (STRUCT) to 60% (DICT) while maintaining 100% precision confirms that the model is not hallucinating alternative values; it is choosing not to extract fields for which it cannot determine the definitive value with sufficient confidence. In a KYC/AML compliance context, this conservative under-extraction is the preferable failure mode. A field left empty triggers a follow-up from the Relationship Manager, while an incorrectly extracted field may propagate silently into the case record.

The MESSY archetype produces $F_1 = 93\%$ with a notably high standard deviation of $\sigma_{F_1} = 14$ and $\sigma_R = 21$. The MESSY prompt embeds the actual client data inside unrelated prose such as sports commentary, restaurant recommendations and weather observations. The high variance reflects the model’s inconsistent ability to separate relevant content from noise across repeated runs. Recall ranges from approximately 68% to 100% across runs of the same prompt, meaning certain fields are extracted on

some runs but missed on others. This behaviour at $T = 0.1$ shows that a near-deterministic temperature setting cannot fully stabilise outputs when the input signal is weak.

TEST2 Analysis: Individual High-Density Account

The TEST2 scenario is based on GT case 551, an individual account for Jamal Kolpataic, a dual-nationality client whose prompts contain the highest field density in the suite: Avaloq risk ratings, passport expiry dates, international phone numbers, AML compliance flags and numerous boolean fields. GPT-5.4 achieves the highest per-scenario consistency of the evaluation in TEST2, with an average F1-score of 97.5% and standard deviations of $\sigma \leq 1.5$ across all four archetypes.

The near-zero standard deviation in TEST2, in sharp contrast to the variance observed in TEST1_MESSY, indicates that the single-party account structure provides sufficiently unambiguous input signals at $T = 0.1$ that the model’s extraction is effectively deterministic. The single-party constraint eliminates the bipartite entity matching problem and the richly labelled STRUCT archetype leaves no room for ambiguity. Even MESSY produces $F_1 = 98\%$ with $\sigma_{F_1} = 0.6$, confirming that GPT-5.4 can reliably filter conversational noise from high-density field prompts.

The sole underperformance in TEST2 is TEST2_EMAIL at $F_1 = 94\%$, where precision drops slightly to 97%. The email format follows the sender’s logical structure rather than any standardised template, meaning some boolean and categorical fields that a Relationship Manager’s notes would explicitly label are expressed as affirmative statements in prose (e.g., “I confirm that I am not a US citizen” rather than `US_person: false`). The minor precision drop suggests that GPT-5.4 occasionally extracts a field from a slightly ambiguous email sentence and assigns a mismatched value. The mismatch penalty rule of the evaluation engine registers this simultaneously as a false positive and a false negative.

TEST3 Analysis: Trust Structure with Seven Entities

The TEST3 scenario is based on GT case 1204, a discretionary revocable trust onboarding for Margrethus AG with seven entities across multiple ontology types. This is the most structurally demanding test case in the suite. The bipartite entity matching problem involves seven instances, several of which share similar names and overlapping role assignments. GPT-5.4 achieves average F1-scores between 98% and 100% across all four archetypes, the strongest validation result in the evaluation.

The near-perfect performance on TEST3 is particularly significant given the per-ontology agent decomposition. Each of the seven entities is handled by a dedicated Ontology Agent operating on a focused sub-prompt. The Orchestrator’s entity matching correctly resolves all seven instances in every archetype, confirmed by the zero false-positive rate across STRUCT, MESSY and DICT. The minor recall shortfalls in TEST3_EMAIL (96%) and TEST3_DICT (98%) follow the same pattern observed in TEST1, where less structured input formats cause a small subset of fields to fall below the model’s extraction confidence threshold, resulting in deliberate under-extraction rather than hallucination.

The cost profile of TEST3 is the highest in the evaluation at approximately \$3.60 per test case, compared to \$1.83 for TEST1 and \$0.94 for TEST2, reflecting the parallel invocation of seven Ontology Agents each processing a non-trivial schema chunk. Despite this, all TEST3 test cases complete within 35–41 s, confirming that the parallel fan-out architecture amortises per-agent latency effectively even under the highest entity load in the suite.

Token Consumption, Cost and Latency Profile

Table 7.6 reports the average input and output token consumption and end-to-end pipeline duration per test case under the GPT-5.4 configuration, computed from raw token metrics recorded by the evaluation framework across all five repeated runs.

Test Case	Input Tokens	Output Tokens	In:Out	Dur (s)
TEST1.STRUCT	716,518	3,428	209 : 1	29.6
TEST1.MESSY	710,690	3,740	190 : 1	31.1
TEST1.DICT	711,883	2,936	242 : 1	35.5
TEST1.EMAIL	709,446	3,940	180 : 1	29.1
TEST2.STRUCT	359,001	2,785	129 : 1	25.2
TEST2.MESSY	353,738	2,836	125 : 1	25.2
TEST2.DICT	360,664	2,221	162 : 1	26.1
TEST2.EMAIL	357,335	3,016	118 : 1	25.2
TEST3.STRUCT	1,413,550	4,213	335 : 1	34.9
TEST3.MESSY	1,413,746	3,752	377 : 1	38.2
TEST3.DICT	1,415,574	3,875	365 : 1	40.8
TEST3.EMAIL	1,415,620	4,123	343 : 1	38.0

Table 7.6: GPT-5.4: input and output token consumption and pipeline duration per test case, each averaged over five repeated runs. In:Out = integer ratio of input to output tokens. Duration reflects the full end-to-end pipeline from prompt submission to confirmed case creation.

Cost scales directly with the number of entity instances the Orchestrator identifies in the prompt, because each instance triggers one parallel set of Ontology Agent invocations, mirroring the input token pattern. The average cost per test case is \$0.94 in TEST2 (one entity instance, one invocation set), \$1.83 in TEST1 (three entity instances, three parallel invocation sets) and \$3.60 in TEST3 (seven entity instances, seven parallel invocation sets). The near-linear progression, TEST1 costs roughly twice TEST2 and TEST3 costs roughly twice TEST1, mirrors the step-wise token doubling visible in Table 7.15. At these price points, GPT-5.4 is the most cost-efficient model in the evaluation. It is cheaper than Sonnet in TEST1 and TEST2, comparable in TEST3 at substantially higher F1 and delivers the highest quality across all three scenarios.

Input token consumption scales directly and predictably with the number of entity instances dispatched in parallel. TEST2, which processes a single entity instance and therefore triggers one set of agent invocations, averages 357,685 input tokens per run. TEST3, which processes seven entity instances in parallel for the trust structure, averages 1,414,622 tokens, a $3.95\times$ increase. The Orchestrator, Process Agent and Prompt Organizer Agent each run once per pipeline execution and consume approximately 130,000–160,000 tokens combined, regardless of whether the case contains one entity or seven. These components handle routing, process selection and prompt restructuring, tasks that do not repeat per entity. Only the Ontology Agent invocations scale with entity count. Because this fixed portion is present in both the TEST2 and TEST3 totals, it dilutes the overall ratio, which is why TEST3 costs roughly four times TEST2 rather than seven times. The extreme input-to-output ratios across all twelve test cases, ranging from 118 : 1 (TEST2.EMAIL) to 377 : 1 (TEST3.MESSY), are a direct consequence of how schema-driven extraction works. Each Ontology Agent receives its full schema window as part of the input: field keys, display names, types, allowed enum values and descriptions for up to 30 fields at a time. This schema documentation alone accounts for the vast majority of input tokens. The output, however, is only the subset of those fields for which matching data was found in the client prompt - a compact JSON dictionary that may contain just a handful of key-value pairs. The ratio therefore measures how much schema the agent had to read per unit of data it extracted. A ratio of 377 : 1 in TEST3.MESSY means the agent consumed 377 input tokens for every single output token it produced, reflecting a case where the schema is large and the matching data sparse. A lower ratio such as 118 : 1 in TEST2.EMAIL indicates more extracted fields relative to the schema size, not a smaller schema.

Pipeline duration follows the same scaling pattern while remaining operationally negligible. TEST2 test cases complete in 25.2–26.1 s across all four archetypes; TEST1 test cases require 29.1–35.5 s; TEST3 test cases peak at 40.8 s for the DICT archetype. Despite processing seven entity instances in parallel, TEST3’s worst-case latency is only 5.3 s above TEST1’s maximum. Because all seven invocations run

simultaneously, the pipeline waits only for the slowest agent to finish rather than accumulating each agent’s latency sequentially (Section 5.7.8). All twelve test cases complete within 41 s, providing a worst-case margin of 79 s to the NFR-1 threshold of 120 s. The DICT archetype consistently produces the longest duration within each scenario. The self-correcting dictation input occasionally causes GPT-5.4 to commit an overridden value; Atfinity rejects it and the self-healing loop resolves it in one pass, adding one correction round-trip to those DICT test cases (see Table 7.13).

7.9.2 Claude Sonnet 4.6: Performance Analysis

Claude Sonnet 4.6 achieves an overall average F1-score of 65.1% across all twelve test cases, a deficit of 31.2 percentage points relative to GPT-5.4, while also costing more per case on average in TEST1 and TEST2. Table 7.7 reports the full result matrix.

Test Case	F1 (%)	σ	Precision (%)	σ	Recall (%)	σ	Cost (\$)	Dur	Ret
TEST1.STRUCT	61	11.0	92	1.8	46	11.5	1.70	50.7	3
TEST1.MESSY	71	16.6	97	4.7	58	18.0	2.16	43.6	1
TEST1.DICT	59	4.7	100	0.0	42	4.6	2.16	40.6	0
TEST1.EMAIL	71	8.6	98	3.0	57	10.6	2.22	42.4	1
TEST2.STRUCT	97	0.7	100	0.0	95	1.4	1.13	36.8	1
TEST2.MESSY	84	14.3	99	1.6	75	20.4	1.19	42.9	2
TEST2.DICT	97	1.4	100	0.0	94	2.6	1.20	42.4	2
TEST2.EMAIL	64	0.0	100	0.0	47	0.0	1.20	41.0	2
TEST3.STRUCT	57	10.9	100	0.0	40	10.4	3.45	48.4	2
TEST3.MESSY	34	11.4	100	0.0	21	8.8	3.41	48.6	2
TEST3.DICT	40	9.0	100	0.0	25	7.4	3.17	44.7	0
TEST3.EMAIL	46	9.7	100	0.0	30	8.4	3.24	45.1	1

Table 7.7: Claude Sonnet 4.6: full evaluation results across all twelve test cases. F1 = F1-score (%); Prec = Precision (%); Rec = Recall (%); σ = standard deviation across repeated runs; Cost = average USD; Dur = average duration in seconds; Ret = total correction cycles per test case across five runs.

The Precision-Recall Imbalance

The defining characteristic of the Sonnet result set is a systematic and severe precision-recall imbalance. Across ten of the twelve test cases, Sonnet achieves precision at or above 98%, including eight cases at exactly 100%, while recall collapses to values between 21% and 95%. Precision at 100% with recall at 21% (TEST3.MESSY) is not a marginal underperformance. The model extracted only 21% of the required fields and left the rest blank.

This imbalance points to an extraction strategy that differs fundamentally from GPT-5.4. The data suggests Sonnet prioritises certainty over completeness. Precision near 100% with severely collapsed recall indicates the model withholds fields (and in more severe cases entire entities) when it encounters semantic ambiguity. The cost data supports this. Sonnet’s average TEST3 cost of \$3.32 is lower than GPT-5.4’s \$3.60 for the same scenario. Because cost scales directly with the number of entity instances processed in parallel, a lower cost implies that fewer Ontology Agent invocations were triggered, meaning Sonnet did not identify all seven trust entities and omitted them from the submission entirely. In a KYC/AML onboarding context, this carries a severe operational risk. A missing field requires manual follow-up; a missing entity means an entire person (a trustee, settlor or beneficiary) is absent from the submitted case and invisible to compliance checks. Figure 7.8 in Section 7.10 visualises the precision-recall imbalance across all three evaluated models and all twelve test cases.

TEST1 Analysis: Recall Degradation Under Multi-Party Load

Sonnet’s TEST1 result is its weakest multi-party result, with an average F1-score of 65.5% and recall values between 42% (DICT) and 58% (MESSY). The high standard deviations, $\sigma_{F_1} = 11.0$ for STRUCT and $\sigma_{F_1} = 16.6$ for MESSY, show that Sonnet’s output varies significantly between repeated runs of the same three-party joint account test.

A counterintuitive pattern appears in the DICT archetype. It produces the lowest recall (42%) but also the lowest standard deviation ($\sigma_{F_1} = 4.7$). This means Sonnet extracts almost the same small set of fields from the dictated-speech input every run - it does not occasionally recover additional fields as GPT-5.4 does. The model has settled into a stable but consistently insufficient extraction behaviour for this input format. It achieves high repeatability at the cost of persistently low recall.

The high precision alongside low recall suggests Sonnet is more selective about which fields it extracts and this selectivity carries a higher cost in TEST1 than in the other scenarios. With three co-holders each contributing fields to the total ground-truth set, the absolute number of fields left blank is larger than in a single-party case, driving down recall and aggregate F1 accordingly.

TEST2 Analysis: Single-Party Peak and Email Boundary

Sonnet’s strongest performance is concentrated in TEST2. The STRUCT and DICT archetypes achieve $F_1 = 97\%$, nearly matching GPT-5.4’s 98–100% range for the same archetypes. This confirms that Sonnet’s extraction capability is not uniformly poor. For a single-party individual account with explicit, well-structured input, it approaches the performance tier of GPT-5.4.

However, TEST2.EMAIL exposes the imbalance sharply. Sonnet achieves $F_1 = 64\%$ with recall of 47% and a standard deviation of zero across all metrics, meaning every run produces identical results. The fields Sonnet does extract are filled in correctly every time - 100% precision confirms this. The problem is not accuracy but coverage. Sonnet consistently extracts the same limited field subset from the email prompt and skips the same remainder on every run. Zero variance rules out stochastic variation; this is a stable, repeatable extraction boundary for this input format.

TEST2.MESSY produces the highest recall standard deviation in the Sonnet result set: $\sigma_{F_1} = 14.3$ and $\sigma_R = 20.4$. Recall swings between roughly 55% and 95% across the five runs, making this the most inconsistent result in the Sonnet evaluation. Noisy, unstructured input causes Sonnet’s extraction to fluctuate significantly from run to run - sometimes capturing most fields, sometimes missing the majority. The same pattern appeared in TEST1.MESSY; the difference here is that only one entity is involved, so the variance is not further amplified across multiple parties.

TEST3 Analysis: Entity Omission and Recall Collapse

TEST3 produces Sonnet’s most severe recall collapse, with an average recall of 29% and an aggregate F1-score of 44.25%, far below the NFR-4 threshold of 85%. Precision remains at 100% across all four archetypes, meaning every extracted field is correct. The problem is how much goes missing.

The primary driver is not just skipped fields within extracted entities but entire entities being omitted. As established in the precision-recall imbalance analysis above, Sonnet’s lower TEST3 cost relative to GPT-5.4 indicates that fewer than seven trust entities were processed. When a trustee, settlor or beneficiary is not identified at all, every field belonging to that person is absent from the submitted case. This compounds the recall collapse far beyond what individual field omissions alone would produce.

TEST3.MESSY is Sonnet’s most extreme result, with $F_1 = 34\%$ and recall of 21%. The standard deviations across TEST3 archetypes (σ_{F_1} ranging from 9.0 to 11.4) show moderate run-to-run variation, but every run remains far below the acceptance threshold.

Token Consumption, Cost and Latency Profile

Table 7.8 reports Sonnet’s token and latency profile. Unlike GPT-5.4, where the token data primarily confirms expected scaling behaviour, the output token dimension here provides additional evidence for the extraction behaviour described in the analysis above.

Test Case	Input Tokens	Output Tokens	In:Out	Dur (s)
TEST1.STRUCT	548,012	3,622	151 : 1	50.7
TEST1.MESSY	699,181	4,163	168 : 1	43.6
TEST1.DICT	701,004	3,618	194 : 1	40.6
TEST1.EMAIL	720,041	3,793	190 : 1	42.4
TEST2.STRUCT	363,860	2,555	142 : 1	36.8
TEST2.MESSY	380,106	3,373	113 : 1	42.9
TEST2.DICT	385,894	2,849	135 : 1	42.4
TEST2.EMAIL	383,497	2,951	130 : 1	41.0
TEST3.STRUCT	1,131,363	4,007	282 : 1	48.4
TEST3.MESSY	1,114,571	4,155	268 : 1	48.6
TEST3.DICT	1,037,126	3,888	267 : 1	44.7
TEST3.EMAIL	1,057,460	4,314	245 : 1	45.1

Table 7.8: Claude Sonnet 4.6: input and output token consumption and pipeline duration per test case, averaged across three to five completed runs depending on the test case. Duration reflects the full end-to-end pipeline latency.

The per-case cost follows entity instance count directly, since each entity triggers one set of parallel Ontology Agent invocations. Running one entity (TEST2) costs \$1.18; three entities (TEST1) costs \$2.06; seven entities (TEST3) costs \$3.32.

TEST2 averages 378,339 input tokens per run; TEST3 averages 1,085,130, a $2.87\times$ increase. Under GPT-5.4, the same step yielded $3.95\times$; Sonnet’s lower ratio reflects the entity omissions documented above, where fewer than seven Ontology Agent invocations ran in TEST3. The fixed pipeline overhead (Orchestrator, Process Agent, Prompt Organizer) contributes a share that does not scale with entity count and is present in both models’ ratios. Input-to-output ratios range from 113 : 1 (TEST2.MESSY) to 282 : 1 (TEST3.STRUCT), driven by each agent receiving a full schema window as input but returning only the matched fields as output.

Despite recall collapsing to 21–40% in TEST3, Sonnet still generates 3,888–4,314 output tokens per test case - a normal extraction volume. The low recall does not come from producing shorter responses; the JSON structure is intact and normally sized, but many of its fields are simply left empty. This confirms that the missing data is a result of fields being skipped, not of the response being cut short.

Pipeline duration ranges from 36.8s (TEST2.STRUCT) to 50.7s (TEST1.STRUCT). The parallel fan-out mechanism keeps total latency bounded to the slowest single agent invocation, so TEST3 - despite up to seven entity invocations running in parallel - does not take seven times as long as TEST2. All twelve test cases complete well within the NFR-1 threshold of 120s. Unlike GPT-5.4, where DICT is the slowest archetype, STRUCT and MESSY produce the longest durations for Sonnet. This is consistent with the retry distribution in Table 7.13: Sonnet accumulates its highest cycle counts in STRUCT and MESSY test cases, where explicit input leads to more submitted fields and more chances for a schema violation. In TEST1 and TEST3, DICT carries zero retries and is the fastest archetype in both.

7.9.3 Claude Haiku 4.5: Performance and Critical Failures

Claude Haiku 4.5 is the least expensive model in the evaluation, with per-case costs of approximately \$0.36 (TEST1), \$0.23 (TEST2, valid case only) and \$0.25 (TEST3), between four and fourteen times cheaper than GPT-5.4 for equivalent test cases. However, this cost advantage is severely undermined by three complete pipeline failures in the TEST2 scenario and by the lowest aggregate F1-score among all models that produced results. Table 7.9 reports the full result matrix.

Test Case	F1 (%)		σ	Precision (%)		σ	Recall (%)		σ	Cost (\$)	Dur	Ret
TEST1.STRUCT	81	10.0		92	3.1		73	13.6		0.38	30.8	2
TEST1.MESSY	82	9.2		91	4.7		75	14.0		0.35	29.7	1
TEST1.DICT	66	7.4		88	1.7		54	9.0		0.35	27.6	0
TEST1.EMAIL	73	17.0		88	4.2		65	20.6		0.36	31.0	2
TEST2.STRUCT	<i>ERROR: LLM pipeline failure; no case created</i>									0	-	25
TEST2.MESSY	<i>ERROR: LLM pipeline failure; no case created</i>									0	-	25
TEST2.DICT	<i>ERROR: LLM pipeline failure; no case created</i>									0	-	25
TEST2.EMAIL	70	14.0		93	3.0		58	18.3		0.23	28.4	4
TEST3.STRUCT	32	3.1		86	1.5		20	2.3		0.26	39.4	4
TEST3.MESSY	24	10.0		77	14.0		14	6.4		0.27	33.1	2
TEST3.DICT	30	3.4		85	1.6		18	2.4		0.26	30.1	1
TEST3.EMAIL	25	2.7		71	2.6		15	1.9		0.23	36.7	3

Table 7.9: Claude Haiku 4.5: full evaluation results. TEST2.STRUCT, TEST2.MESSY and TEST2.DICT produced complete LLM errors; no case was created and all metrics are zero. F1 = F1-score (%); Prec = Precision (%); Rec = Recall (%); σ = standard deviation; Cost = average USD; Dur = average duration in seconds; Ret = total correction cycles per test case across five runs (25 = correction budget exhausted on every run).

TEST1 Analysis: Joint Account, Three Co-Holders

Haiku achieves an average F1-score of 75.5% across the four TEST1 archetypes, with STRUCT and MESSY as the strongest results (81% and 82%) and DICT as the weakest (66%). Standard deviations are notably high across all four archetypes: $\sigma_{F_1} = 10.0$ for STRUCT, 9.2 for MESSY, 7.4 for DICT and 17.0 for EMAIL, indicating significant run-to-run variability in Haiku’s TEST1 output.

Haiku’s precision values across TEST1 (88–92%) indicate that some extracted field values are incorrect. The model attempts to fill fields even where the mapping is not fully certain, suggesting a task-completion orientation. It submits the field values it can map from the prompt rather than withholding uncertain assignments. Its recall (54–75%), while substantially above its TEST3 performance, shows that a meaningful portion of ground-truth fields still goes unextracted.

Three of the four TEST1 archetypes triggered Atfinity’s self-healing correction loop: STRUCT accumulated 2 correction cycles in total across five runs, MESSY 1 and EMAIL 2. This means Haiku submitted payloads containing field values that Atfinity’s validation layer rejected before case creation could proceed. The retries are a direct consequence of the broader extraction pattern. Committing uncertain field values to the payload increases the likelihood that some are rejected by Atfinity’s validator and require a correction cycle.

The TEST1.EMAIL archetype produces Haiku’s highest standard deviation in this scenario: $\sigma_{F_1} = 17.0$ and $\sigma_R = 20.6$. Recall fluctuates between approximately 45% and 85% across repeated runs, making this the most variable result in the TEST1 group. Email-format inputs lack the explicit field labelling that structured or dictated prompts provide and this ambiguity translates directly into extraction instability at $T = 0.1$.

TEST2 Failures: Critical Reliability Gap

Three of the four TEST2 archetypes produced complete LLM pipeline failures: `test2_struct`, `test2_messy` and `test2_dict` returned zero across all metrics, with no case created in Atfinity. Only `test2_email` produced a valid result ($F_1 = 70\%$, recall = 58%). This failure pattern is not a performance degradation but a hard reliability boundary that disqualifies Haiku from any production deployment scenario involving this class of high-density individual account cases.

GT case 551 contains the highest number of populated fields in the evaluation suite: Avaloq risk ratings, passport data, international phone numbers, AML compliance flags and boolean indicators. Many of these are strongly typed fields with strict format constraints. When Haiku extracts a value with the wrong format, Atfinity rejects it and the self-healing loop routes the error back to Haiku. Haiku then fails to produce a valid correction. If Atfinity requires a decimal scalar with two decimal places and Haiku submitted a plain integer, it returns 0 instead of 0.00; the type constraint remains unsatisfied and the retry is consumed without progress. This mirrors the GPT-5.4-mini failure mode documented in Section 7.9.4. After five attempts the self-healing budget is exhausted and no case is created.

The TEST2_EMAIL result is not an exception to this pattern but a consequence of it. Email prose does not present typed values explicitly. A sentence like “his income is around 100 dollars” does not give Haiku enough signal to map the value to a strongly typed decimal field reliably. Haiku therefore skips those fields entirely rather than attempting an extraction that would fail validation. The result is a payload that reaches Atfinity without type errors, but with only 58% of the required fields populated. The 58% recall is the direct evidence: the missing fields are not withheld by choice but were never extracted. The failure in STRUCT, MESSY and DICT is the same underlying incapability made visible. Those archetypes state the values explicitly, Haiku extracts them with incorrect formatting and the retry loop cannot recover.

The cost entries for the three failed archetypes show \$0.00 in Table 7.9. This does not mean the runs consumed no API tokens. The cost logger runs after successful case submission. When a run terminates with a pipeline exception before that point, no cost is written to the results file. This is a known framework limitation; the corrected behavior and its implications are discussed in Section 7.11.

TEST3 Analysis: Compound Failure Across Seven Agents

Haiku’s TEST3 results are its weakest in the evaluation: average F1-scores of 24–32% and average recall of 14–20%. The cost data reveals the primary driver of the recall collapse. TEST3 costs \$0.23–\$0.27 per test case, barely above TEST2_EMAIL at \$0.23, despite the trust case requiring seven simultaneous Ontology Agent invocations. Since cost scales directly with the number of entity instances the Orchestrator identifies and dispatches, these figures indicate that Haiku processes only one or at most two of the seven trust entities. The remaining entities are never dispatched and every field belonging to them is absent from the submitted payload. For reference, TEST1 already costs \$0.35–\$0.38 for three entities; a correctly dispatched seven-entity case should cost substantially more. Entity omission is therefore the primary driver of the recall collapse, not field-level extraction failures within processed entities.

For the entities Haiku does process, precision also fails: 71–86% across the four archetypes, compared to 88–92% in TEST1. With seven entities sharing overlapping role vocabularies, Haiku’s task-completion approach produces cross-entity field attribution errors. A value belonging to one role-holder is mapped to the schema slot of another. Atfinity’s validator does not detect semantic misassignments - it only rejects format violations - so these incorrect values pass through without triggering a retry. This is consistent with the retry counts of one to four across all four TEST3 archetypes. TEST3_MESSY is the most severe case, with 77% precision and $\sigma_P = 14$, the highest precision standard deviation in the evaluation. One in four extracted field values is incorrect and the high σ_P shows the specific errors vary across runs rather than following a predictable pattern.

STRUCT and DICT show $\sigma_{F_1} = 3.1$ and 3.4, among the lowest standard deviations in the Haiku result set. The extraction output is nearly identical across all five runs but consistently insufficient. A small subset of fields from a subset of entities is produced repeatedly, while the majority of the ground-truth payload - both missing entities and misattributed fields - remains absent.

Token Consumption, Cost and Latency Profile

Table 7.10 documents Haiku’s token and latency profile. The input token and cost data independently confirm the entity omission and extraction behaviour identified in the preceding sections.

Test Case	Input Tokens	Output Tokens	In:Out	Dur (s)
TEST1.STRUCT	446,246	6,480	69 : 1	30.8
TEST1.MESSY	411,885	6,109	67 : 1	29.7
TEST1.DICT	405,386	5,663	72 : 1	27.6
TEST1.EMAIL	413,424	6,139	67 : 1	31.0
TEST2.STRUCT	<i>ERROR: LLM pipeline failure</i>			
TEST2.MESSY	<i>ERROR: LLM pipeline failure</i>			
TEST2.DICT	<i>ERROR: LLM pipeline failure</i>			
TEST2.EMAIL	265,678	4,208	63 : 1	28.4
TEST3.STRUCT	298,446	4,665	64 : 1	39.4
TEST3.MESSY	318,411	4,411	72 : 1	33.1
TEST3.DICT	300,314	3,800	79 : 1	30.1
TEST3.EMAIL	257,832	4,968	52 : 1	36.7

Table 7.10: Claude Haiku 4.5: input and output token consumption and pipeline duration per test case, averaged over five repeated runs. The three TEST2 error rows log no token metrics as the pipeline failed before Ontology Agent completion.

The most significant finding in the input token dimension is the TEST3 figure. TEST3 averages 293,751 input tokens per run, which is lower than TEST1’s 419,235, despite TEST3 involving seven entity instances and TEST1 only three. If all seven trust entities were dispatched, input tokens should be roughly double the TEST1 value. Instead they are 30% lower and barely above TEST2_EMAIL’s 265,678, a single-entity case. This is the token-level confirmation of the entity omission documented in the TEST3 analysis. Only one or at most two Ontology Agent invocations ran per TEST3 execution. The remaining role-holders were never dispatched and their tokens were never consumed.

Output tokens are high relative to input, with input-to-output ratios of 52–79 : 1. The primary driver is that Haiku extracts and outputs more field values per invocation rather than leaving uncertain fields blank. Every extracted value - correct or not - contributes to output volume. The self-healing retries add a secondary contribution; each correction cycle generates additional output tokens for the corrected values. In TEST1, output tokens range from 5,663 to 6,480 per test case. In TEST3, they fall to 3,800–4,968, lower than TEST1 despite the trust case nominally involving more entities. The drop is consistent with fewer agents having run, since less total output is produced when fewer invocations complete.

Per-case cost is \$0.35–\$0.38 for TEST1, \$0.23 for TEST2.EMAIL and \$0.23–\$0.27 for TEST3. The TEST3 range, barely above the single-entity TEST2.EMAIL figure, is the cost-dimension confirmation of the input token finding. The three failed TEST2 archetypes show \$0.00 in Table 7.9, a consequence of the framework logging limitation described above rather than an actual zero API cost.

Pipeline duration for completed test cases ranges from 27.6 s (TEST1.DICT) to 39.4 s (TEST3.STRUCT). The small latency gap between TEST1 and TEST3 (3 to 9 s) is a further signal that few additional parallel invocations ran in TEST3. If seven agents had run in parallel, the duration would be bounded by the slowest of the seven; with only one or two agents running, the fan-out provides no meaningful speedup over TEST1. All completed test cases fall well within the NFR-1 threshold of 120 s.

7.9.4 GPT-5.4-mini: Complete Pipeline Rejection

GPT-5.4-mini was evaluated as the fourth candidate model. Unlike the three models that produced complete or partial result sets, GPT-5.4-mini failed at the pipeline level. The model was unable to correctly map extracted field values to Atfinity’s field schema in a way that satisfied the platform’s validation layer, producing payloads that were rejected in full on nearly every test case. Because no case was successfully created in Atfinity during the GPT-5.4-mini evaluation runs, the deterministic evaluation engine had no created case to retrieve and score against the ground truth. The result is not a low F1-score but the complete absence of any evaluable output.

Smaller models in the same family tend to follow output format instructions less reliably. GPT-5.4 returns plain scalar values as required. GPT-5.4-mini wraps the same values in a nested object instead, ignoring the format constraint. The failure mechanism shares the same root cause as the Haiku TEST2 failures documented in Section 7.9.3. The model cannot correctly resolve Atfinity’s field validation errors when they are routed back through the self-healing loop. GPT-5.4-mini produces structurally valid JSON, but the submitted values violate Atfinity’s field type constraints in a recurring pattern. Atfinity rejects each payload in its entirety, consistent with its atomic POST semantics, where one invalid field causes the full submission to be refused. The self-healing loop then routes each rejected field back to the responsible Ontology Agent’s `/correct` endpoint. GPT-5.4-mini cannot produce a valid correction. If Atfinity requires a plain decimal scalar and the model submitted a nested JSON object, the correction attempt returns an integer rather than the required decimal format. The type constraint remains unsatisfied, the retry is consumed without progress and after five attempts the self-healing budget is exhausted. Figures 7.5 and 7.6 show the raw pipeline logs from a representative run, documenting the rejection cycle and the failed correction attempts.

```
[API] POST https://ost.atfinity.app/api/1/cases → 400

Atfinity rejection detail:
Field : total_wealth_bankable_assets_amount_chf
Error : conversion from dict to Decimal is not supported
Value : {'decimal_number': 0.0, 'unit': 'chf', 'item': None}
          ~~~~~
          GPT-5.4-mini submitted a nested JSON object.
          Atfinity requires a plain decimal scalar.

[ORCH] Step 5 | Attempt 5/5 - Atfinity rejected the payload.
          All retry attempts exhausted. Case creation failed.
```

Figure 7.5: Atfinity’s HTTP 400 rejection on the fifth and final retry attempt for a GPT-5.4-mini evaluation run. The model consistently formatted currency-amount fields as nested JSON objects containing `decimal_number`, `unit` and `item` keys rather than the plain decimal scalar that Atfinity’s API requires. This type mismatch triggered a rejection on every submission attempt.

```

-- Correction attempt 1 --
[Person] Field to correct : total_income_others_estimate_chf
        Submitted value  : {'decimal_number': 0.0, 'unit': 'chf', 'item': None}
        Atfinity error    : conversion from dict to Decimal is not supported
[LLM]    Model           : gpt-5.4-mini
        Corrected value   : None          ← field dropped instead of corrected

-- Correction attempt 2 --
[Person] Field to correct : total_wealth_bankable_assets_amount_chf
        Submitted value  : {'decimal_number': 0.0, 'unit': 'chf', 'item': None}
        Atfinity error    : conversion from dict to Decimal is not supported
[LLM]    Model           : gpt-5.4-mini
        Corrected value   : 0             ← integer returned, not a valid decimal

[API]    POST .../cases → 400 (attempt 5/5, all retries exhausted)

-----

-- Same test case, GPT-5.4 (non-mini) --
[API]    POST .../cases → 200 (attempt 1/5)
[ORCH]   case_id=1289, 90 field(s) set, 4 instance(s) created

```

Figure 7.6: Self-healing loop execution for GPT-5.4-mini alongside the contrasting GPT-5.4 result for the identical test case. When the Orchestrator routed the rejected fields back to the Person Agent with the full error context, GPT-5.4-mini returned `None` for the first field (stripping it rather than correcting it) and an integer `0` for the second, neither satisfying Atfinity’s type constraint. GPT-5.4, given the same prompt and schema, required no correction at all and succeeded on its first submission attempt.

The GPT-5.4-mini failure carries a direct implication for model selection. A smaller model from the same provider family does not inherit the schema-mapping capability of a larger one. The gap between GPT-5.4 and GPT-5.4-mini is not a quantitative accuracy degradation but a qualitative capability boundary. GPT-5.4-mini cannot reliably interpret and follow the structured field-mapping instructions that the Ontology Agent prompt requires. This reinforces the conservative model selection guidance of Section 7.7.3. Cost optimisation through model downsizing must be validated empirically against full Atfinity payload compliance before any deployment. General benchmark scores alone are not a sufficient indicator of schema-mapping capability for this use case.

7.10 Cross-Model Comparison

This section aggregates the results from all three evaluated model configurations. It opens with a scenario-level F1 overview, then breaks down precision and recall profiles, NFR compliance and operational metrics.

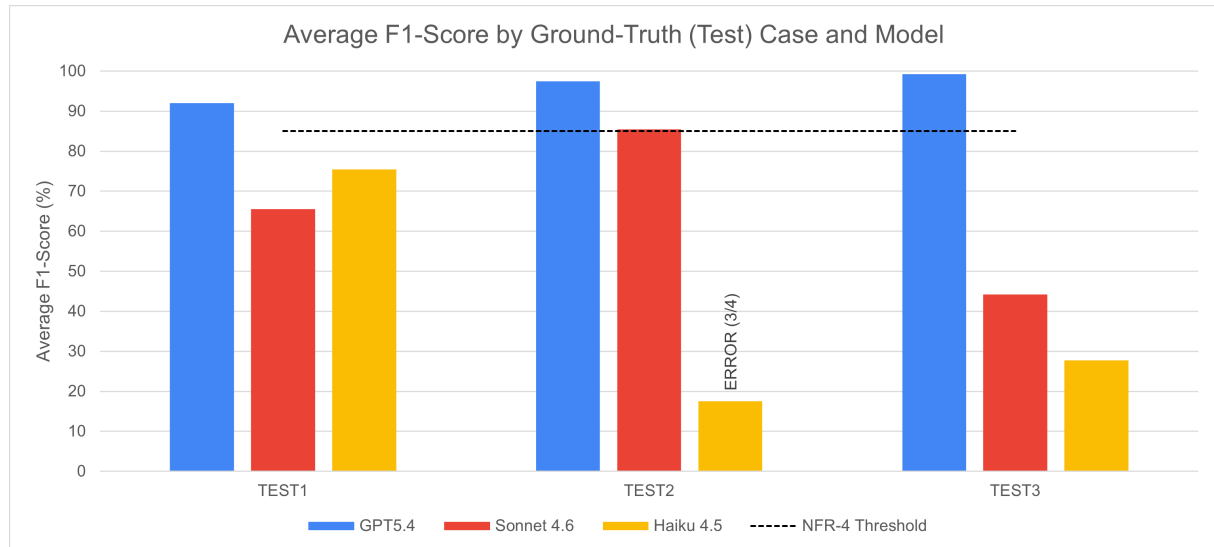


Figure 7.7: Average F1-score per ground-truth scenario for each evaluated model, with each bar representing the mean across the four conversational archetypes (Structured, Messy, Dictation and Email). The dashed line marks the NFR-4 acceptance threshold of 85%.

Figure 7.7 collapses the four conversational archetypes into a single average bar per model and scenario, making the cross-model and cross-scenario performance gradient immediately visible. GPT-5.4 is the only configuration that clears the NFR-4 threshold in all three scenarios, with scenario averages of 92% (TEST1), 97.5% (TEST2) and 99.25% (TEST3). That GPT-5.4 reaches its highest average on the most structurally complex scenario, TEST3, the trust case with seven entities, rather than the simplest is counterintuitive but explicable. Both scenarios share the same four archetypes, including DICT. The difference lies in how the DICT archetype interacts with each scenario’s content: TEST1.DICT scores 75% because dictated joint-account input leaves co-holder role assignments ambiguous across three parties, reducing recall for role-specific fields; TEST3.DICT scores 99% because trust structures employ formal, unambiguous role vocabulary - trustee, settlor, beneficiary - that the model maps correctly even from transcribed speech. The lower TEST1 average is therefore a scenario-specific vulnerability within the DICT archetype, not a general difficulty ordering between scenarios. Across the remaining archetypes, GPT-5.4’s performance is effectively saturated.

Sonnet’s bars reveal a clear entity-count dependency. TEST2 is Sonnet’s strongest scenario at an average of 85.5%, narrowly crossing the NFR-4 line, driven by near-perfect STRUCT and DICT archetypes ($F_1 = 97\%$ each). TEST1 drops to 65.5% as the three-party joint account forces the model to map role-specific fields correctly across multiple co-holders, a task where recall degrades significantly. TEST3 collapses to 44.25%. Sonnet’s recall falls to an average of 29% across the seven trust entities, while precision remains at 100%. The model extracts correctly when it acts, but it increasingly omits the full ground-truth field set as entity count grows.

The Haiku bars require a different reading entirely: the Haiku TEST2 bars are not low-scoring results but the absence of results. Three of the four TEST2 archetypes exhausted the five-attempt self-healing budget on every run and the single visible TEST2 bar derives from the EMAIL archetype alone ($F_1 = 70\%$). The apparent Haiku TEST1 outperformance relative to Sonnet (75.5% versus 65.5%) is a known anomaly addressed in Section 7.9.3; it does not reflect a genuine hierarchy inversion but a scenario-specific difference in how each model handles the three-party joint account input.

Table 7.11 presents the aggregate precision and recall metrics for the three models, averaged across

all test cases for which valid results are available.

Model	Avg Precision (%)	Avg Recall (%)
GPT-5.4	99.6	94.3
Sonnet 4.6	98.9	52.1
Haiku 4.5	84.7	39.1

Table 7.11: Average precision and recall per model across all twelve test cases.

The aggregate comparison reveals three distinct performance tiers. GPT-5.4 occupies the top tier alone, with an average F1-score that exceeds the NFR-4 threshold of 85% in all three test scenarios. This is counter to the general capability expectation: Sonnet 4.6 is a newer and larger model than GPT-5.4 and would ordinarily be expected to lead. In this evaluation it does not, for reasons established in Section 7.9.2. Sonnet occupies a middle tier. It meets the NFR-4 threshold in TEST2 (85.5%) but fails substantially in TEST1 and TEST3. Haiku occupies the bottom tier for TEST2 and TEST3, with one notable exception. It outperforms Sonnet in TEST1, the only scenario in the evaluation where the expected model capability hierarchy is inverted.

A practical question is whether a smaller or cheaper model produces equivalent results. Haiku’s failures in TEST2 and TEST3 show that the smallest model evaluated is not sufficient. However, size and cost alone do not predict success. Sonnet is a large model that costs as much as or more than GPT-5.4 in TEST1 and TEST2, yet it also fails the recall requirement in all but the single-party scenario. Of the three models evaluated, only GPT-5.4 meets all four NFRs across all scenarios. The determining factor is not model size or cost but the ability to perform reliable schema-conditioned extraction under multi-party load, a property only GPT-5.4 demonstrates consistently in this evaluation.

The characteristic failure mode differs across all three models. GPT-5.4’s recall collapses only under DICT temporal self-corrections and resolves within a single retry cycle, leaving no lasting impact on precision or F1. Sonnet’s primary failure is entity identification. The Orchestrator does not reliably extract all role-holders from multi-party input, so entire persons are absent from the submitted case rather than individual fields being missed. Haiku’s primary failure is schema type compliance. It submits constrained field values in an incompatible format that the self-healing loop cannot correct within the five-cycle budget, producing complete pipeline failures rather than low-recall submissions.

Table 7.12 maps each model against the four quantitative non-functional requirements.

NFR	Threshold	GPT-5.4	Sonnet 4.6	Haiku 4.5
NFR-1	Latency ≤ 120 s	Met (max: 40.8 s)	Met (max: 50.7 s)	Met (max: 39.4 s)
NFR-2	Precision $\geq 90\%$	Met (avg: 99.6%)	Met (avg: 98.9%)	Not met (avg: 84.7%)
NFR-3	Recall $\geq 90\%$	Met (avg: 94.3%)	Not met (avg: 52.1%)	Not met (avg: 39.1%)
NFR-4	F1-Score $\geq 85\%$	Met (avg: 96.3%)	Partial (TEST2 only: 85.5%)	Not met (avg: 40.3%)

Table 7.12: NFR compliance assessment per evaluated model. Aggregate averages are reported for context; individual test case values are available in Tables 7.5–7.9.

GPT-5.4 is the only model that meets all four quantitative NFRs. NFR-3 (recall $\geq 90\%$) is the binding constraint that eliminates both Sonnet and Haiku from full compliance. Aggregate recall values of 52.1% and 39.1% respectively are far below the minimum required for a system that must reliably populate compliance-required KYC/AML fields. NFR-1 (latency ≤ 120 s) is met by all three models without exception. GPT-5.4 is therefore the only configuration that satisfies all defined quality, reliability and latency thresholds for bank client onboarding. Claude Sonnet 4.6 is viable in the single-party scenario

(TEST2: 85.5% F1) but fails the recall requirement for multi-party and trust cases. Claude Haiku 4.5 fails NFR-2 through NFR-4 in TEST3 and fails categorically in three of four TEST2 archetypes.

7.10.1 Precision and Recall Profile

The recall dimension is the primary differentiator between GPT-5.4 and the other two models. GPT-5.4 achieves 94.3% average recall, meeting the NFR-3 threshold of 90% at the aggregate level. Sonnet achieves 52.1% and Haiku 39.1% (including errors), both far below the 90% NFR-3 minimum. Precision is high across all three models, GPT-5.4 at 99.6%, Sonnet at 98.9% and Haiku at 84.7%. GPT-5.4 and Sonnet meet the NFR-2 precision threshold of 90% at the aggregate level; Haiku does not. The binding NFR failure for Sonnet and Haiku is recall, not precision. Figure 7.8 plots precision against recall for every individual test case and model, making the spatial separation between models immediately visible.

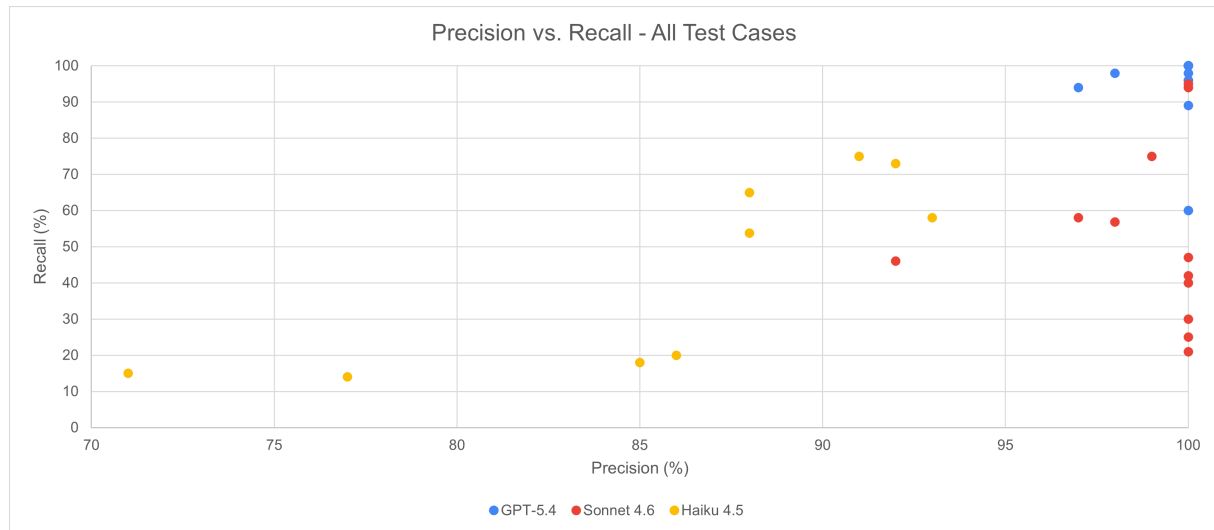


Figure 7.8: Precision-recall scatter plot per model and test case across the full evaluation suite. Each point represents one of the twelve test cases per model. The three Haiku TEST2 pipeline failures are excluded as their values fall outside the plotted range.

Figure 7.8 makes the model separation immediately visible. GPT-5.4 clusters in the top-right region, with both precision and recall consistently above 89% across eleven of twelve test cases, with TEST1_DICT the lone exception at 60% recall. Sonnet's points align along the right edge: precision at or near 100%, but recall collapsing to as low as 21%, confirming the systematic imbalance documented in Section 7.9.2. Haiku shows the widest spread. The four TEST3 data points cluster in the bottom-left region, with precision between 71–86% and recall between 14–20%. These are the only points in the evaluation that fall below the NFR-2 precision threshold of 90%. The precision drop in TEST3 is caused by cross-entity field attribution errors. With seven entities sharing overlapping role vocabularies, Haiku maps values from one entity's schema slot to another and Atfinity does not detect semantic misassignments (see Section 7.9.3). The spatial separation between the three clusters mirrors the three performance tiers identified in the aggregate comparison above.

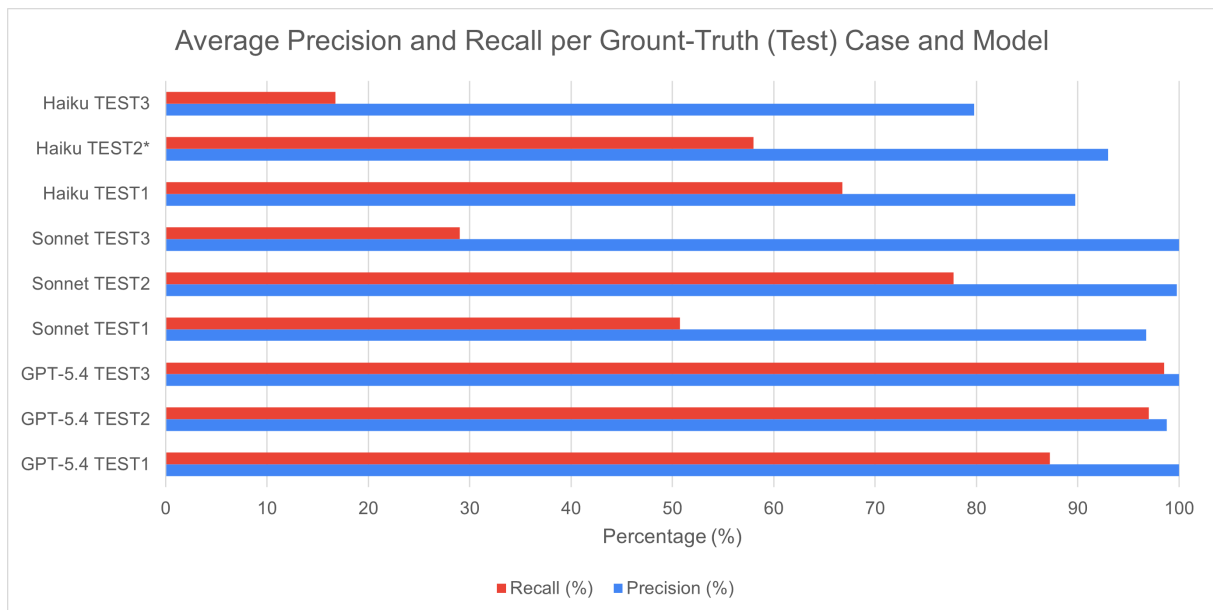


Figure 7.9: Average precision and recall per model and ground-truth scenario, where each bar represents the mean across all conversational archetypes within that scenario. Haiku TEST2* includes the email archetype only; the three pipeline failures are excluded. Complements Figure 7.8: where the scatter plot shows one data point per archetype (e.g. TEST1.STRUCT, TEST1.DICT), this chart collapses those points into a single scenario-level average, making the per-scenario trend directly readable.

Figure 7.9 is complementary to the scatter plot: it averages the individual archetype points from Figure 7.8 into one precision and one recall value per scenario, shifting the focus from intra-scenario spread to the overall scenario-level direction. GPT-5.4’s recall drops from 96–99% in TEST2 and TEST3 to 87% in TEST1. The cause is the DICT archetype: its stream-of-consciousness input with mid-sentence self-corrections causes the model to skip fields where the intended value is ambiguous, pulling recall down from 100% to 60% for that archetype alone (see Section 7.9.1). Sonnet’s recall declines monotonically with scenario complexity: 77% in TEST2, 51% in TEST1 and 29% in TEST3. The driver is entity omission: the Orchestrator’s failure to identify all role-holders in multi-party input worsens as the number of entities grows, from one in TEST2 to three in TEST1 to seven in TEST3. Precision stays at or above 97% across all three because when Sonnet does dispatch an agent and extract a field, the value is correct; the failure is in what is omitted, not in what is submitted (see Section 7.9.2). Haiku TEST3 is the only scenario where precision falls below the NFR-2 threshold of 90%, reaching 80%; its recall of 17% is also the lowest recorded across any successful scenario in the evaluation.

TEST1 Hierarchy Inversion: Haiku and Sonnet

Haiku achieves 75.5% average F1 in TEST1 while Sonnet achieves only 65.5% on the same scenario. The primary driver is entity identification. TEST1 presents three co-holders, Peter, Paul and Gerbert Müller, each assigned multiple roles. The data suggests that Sonnet’s Orchestrator does not reliably identify all three persons from the input. When a co-holder is not recognised, no Ontology Agent is dispatched for that person and every field belonging to them is absent from the submitted case. With three co-holders each contributing roughly a third of the ground-truth field set, missing even one person collapses recall substantially. Haiku appears to identify the co-holders more consistently. It dispatches agents for all three and submits every field it can match, including some that turn out to be incorrect. The retry data in Table 7.9 supports this. Three of the four TEST1 archetypes triggered Atfinity’s self-healing correction loop: STRUCT accumulated 2 correction cycles in total across five runs, MESSY 1 and EMAIL 2. These retries confirm that Haiku submitted values that Atfinity’s validator rejected, values that Sonnet would not have submitted at all because the corresponding entities were never processed. This outcome is scenario-specific. The three-entity structure of TEST1 amplifies Sonnet’s entity identification weakness in a way that single-party cases do not. It does not reflect a genuine reversal of model capability.

A consistent input format pattern holds across all three models. Figure 7.10 shows the full F1 heatmap across all twelve test cases and models.

	GPT-5.4	Sonnet 4.6	Haiku 4.5
TEST1_STRUCT	100% $\pm$ 0.0	61% $\pm$ 11.0	81% $\pm$ 10.0
TEST1_MESSY	93% $\pm$ 14.0	71% $\pm$ 16.6	82% $\pm$ 9.2
TEST1_DICT	75% $\pm$ 4.5	59% $\pm$ 4.7	66% $\pm$ 7.4
TEST1_EMAIL	100% $\pm$ 0.0	71% $\pm$ 8.6	73% $\pm$ 17.0
TEST2_STRUCT	98% $\pm$ 1.2	97% $\pm$ 0.7	ERROR
TEST2_MESSY	98% $\pm$ 0.6	84% $\pm$ 14.3	ERROR
TEST2_DICT	100% $\pm$ 0.0	97% $\pm$ 1.4	ERROR
TEST2_EMAIL	94% $\pm$ 0.6	64% $\pm$ 0.0	70% $\pm$ 14.0
TEST3_STRUCT	100% $\pm$ 0.0	57% $\pm$ 10.9	32% $\pm$ 3.1
TEST3_MESSY	100% $\pm$ 0.0	34% $\pm$ 11.4	24% $\pm$ 10.0
TEST3_DICT	99% $\pm$ 1.6	40% $\pm$ 9.0	30% $\pm$ 3.4
TEST3_EMAIL	98% $\pm$ 4.26	46% $\pm$ 9.7	25% $\pm$ 2.7

Figure 7.10: Mean F1-score and run-to-run standard deviation per test case across all three evaluated models.

Each cell reports the mean F1-score and run-to-run standard deviation ($\pm\sigma$) over five repeated runs. Green cells meet the NFR-4 threshold ($\geq 85\%$ F1); red cells do not; ERROR marks a complete pipeline failure. STRUCT produces the highest F1 for every model. DICT is among the weakest archetypes across all three models. The temporal self-correction challenge, where a speaker immediately overrides a value mid-sentence, consistently reduces F1 regardless of model. MESSY and EMAIL both produce the highest standard deviations. The source of variance is the input itself, not the model’s random sampling. At $T = 0.1$, a near-deterministic temperature cannot fully stabilise outputs when the input signal is weak. The model’s ability to separate relevant content from noise varies across repeated runs of the same prompt, as established in Section 7.9.1. Smaller models are affected more severely because they are less consistent at filtering noise from complex unstructured input, amplifying the variance that all models exhibit on these archetypes. The differential is clearest at the DICT archetype: GPT-5.4’s lowest F1 across all DICT test cases is 75% (TEST1_DICT); Sonnet reaches 40% (TEST3_DICT) and Haiku 30% (TEST3_DICT). All three models degrade from STRUCT to DICT, but the absolute floor remains substantially higher for GPT-5.4. The reason is resolution capability: when a speaker immediately overrides a value, a more capable model can determine which of the two stated values was intended. Smaller models are less reliable at this disambiguation and tend to either omit the field or commit the wrong value. The format sensitivity does not impose an operational constraint on the Relationship Manager. The Prompt Organizer Agent (ADR11, Section 5.7.11) transforms the RM’s free-form input into clearly labelled, entity-separated sections before extraction, without adding, inferring or removing content. The format gradient documents the design target of the Prompt Organizer Agent, not a requirement placed on the RM.

7.10.2 Reliability and Schema Compliance

The self-healing retry mechanism defined in FR10 is triggered each time Atfinity’s API returns a non-2xx HTTP response code. Each correction cycle routes the rejected field to the responsible Ontology Agent’s `/correct` endpoint, applies a correction and resubmits the payload. The maximum budget is five cycles; exhausting it constitutes a pipeline failure. Figure 7.11 shows the total number of correction cycles per test case and model, summed across all five evaluation runs.

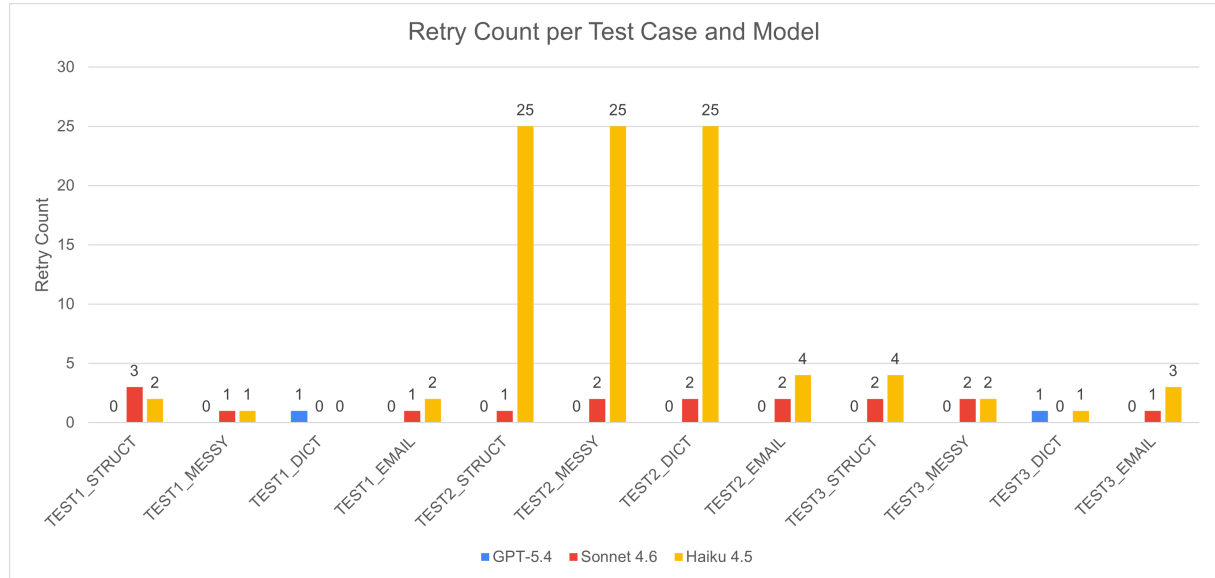


Figure 7.11: Total correction cycles per test case and model, summed over five independent runs.

A value of zero means no correction cycle was triggered across any of the five runs. The maximum observable value is 25, which represents the five-cycle budget exhausted on every run.

GPT-5.4 records zero correction cycles in ten of twelve test cases. TEST1.DICT and TEST3.DICT each accumulate a single correction cycle across all five runs. The loop was triggered exactly once throughout the entire evaluation for each case. GPT-5.4 occasionally commits the overridden value from a self-correcting DICT input, Atfinity rejects it and the loop resolves it in one pass. The effect on precision and duration is negligible.

Sonnet accumulates between one and three total correction cycles in ten of twelve test cases; TEST1.DICT and TEST3.DICT require none. The correction budget is never approached - the maximum across all test cases is three total cycles, well within the 25-cycle ceiling. Precision is maintained: rejected fields are corrected and re-submitted successfully. The loop does not improve recall, since it acts only on fields already present in the payload; fields omitted due to entity extraction failure are absent entirely and no correction cycle can recover them.

Haiku’s failure mode is the most severe. TEST2.STRUCT, TEST2.MESSY and TEST2.DICT each accumulate 25 total correction cycles, meaning the five-cycle budget was exhausted in every one of the five runs. The cause is type constraint failure: Haiku submits constrained field values in an incompatible format and the loop cannot produce a schema-valid replacement within the per-run budget. These three cases produce zero precision and zero recall. TEST2.EMAIL accumulates four total correction cycles across five runs and succeeds in each, because email prose does not state typed field values explicitly, causing Haiku to skip uncertain fields rather than submit an incorrect format. Outside TEST2, correction cycles range from zero to four total, reflecting sporadic but recoverable schema violations.

Figure 7.12 shows end-to-end pipeline duration per test case. Table 7.13 lists the retry counts that drive it.

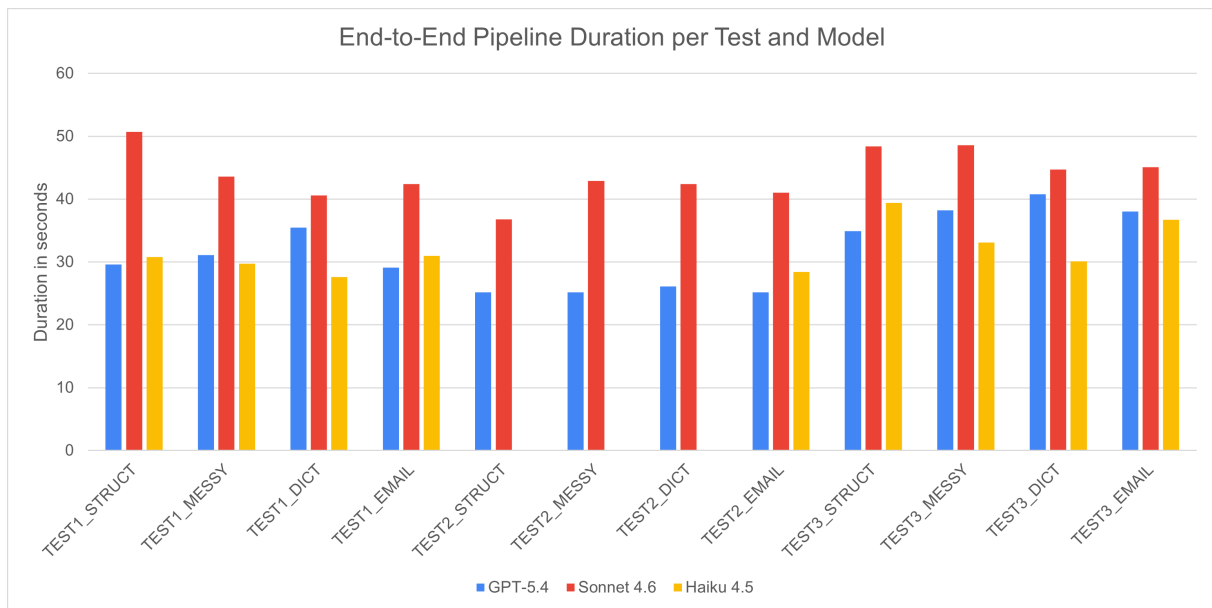


Figure 7.12: End-to-end pipeline duration per test case and model, grouped by ground-truth scenario.

Test Case	GPT-5.4	Sonnet 4.6	Haiku 4.5
TEST1.STRUCT	0	3	2
TEST1.MESSY	0	1	1
TEST1.DICT	1	0	0
TEST1.EMAIL	0	1	2
TEST2.STRUCT	0	1	25
TEST2.MESSY	0	2	25
TEST2.DICT	0	2	25
TEST2.EMAIL	0	2	4
TEST3.STRUCT	0	2	4
TEST3.MESSY	0	2	2
TEST3.DICT	1	0	1
TEST3.EMAIL	0	1	3

Table 7.13: Total correction cycles per test case and model, summed over five independent runs.

Retry count is the primary driver of duration. Each retry adds a correction inference and an Atfinity round-trip. For GPT-5.4, DICT carries the highest retry count and is consistently the slowest archetype. For Sonnet, DICT has zero retries in TEST1 and TEST3 and is the shortest archetype in both. This is consistent with Sonnet’s conservative extraction strategy: the ambiguous self-correcting input of the DICT archetype causes Sonnet to omit uncertain fields entirely rather than risk submitting an incorrect value. An omitted field does not reach Atfinity and therefore triggers no validation rejection and no correction cycle. STRUCT states values explicitly, giving Sonnet more fields to extract; more submitted fields create more chances for a schema violation and correspondingly higher retry counts. STRUCT therefore carries the highest cycle count in TEST1 and TEST3 and is consistently the longest archetype in both.

The retry distribution reflects a broader pattern across model scales. Larger models tend to be more calibrated. When a field mapping is uncertain, Sonnet withholds the field rather than guessing. The goal is a correct submission, not a complete one. Smaller models behave differently. Haiku attempts to map

and submit every field it can interpret, even when the confidence is low. The goal is task completion. The result is more submissions, more Atfinity rejections and more correction cycles. Haiku’s TEST1 retries are a direct consequence of this orientation. Sonnet’s zero DICT retries are a direct consequence of the opposite. Neither approach is unconditionally better: calibration raises precision but lowers recall; task-completion raises recall but increases schema violations. For Haiku, the retry ordering within each scenario maps directly onto the duration ordering: DICT accumulates the fewest cycles and is consistently the fastest archetype, while STRUCT accumulates the most and is the slowest. Haiku’s TEST2 failures each consume five correction round-trips per run before aborting, making them the longest-running cases in the evaluation. All successful test cases complete well within the NFR-1 threshold of 120 s.

7.10.3 Cost and Operational Analysis

This section examines the cost, operational efficiency and token consumption of each evaluated model. Cost is assessed against extraction quality to determine whether higher spend translates to higher F1. Operational metrics capture throughput and latency characteristics. Token consumption traces how input and output volume scale across scenarios of increasing complexity.

Cost and Quality Trade-off

Figure 7.13 plots the average cost per case against the average F1-score for each model and ground-truth scenario.

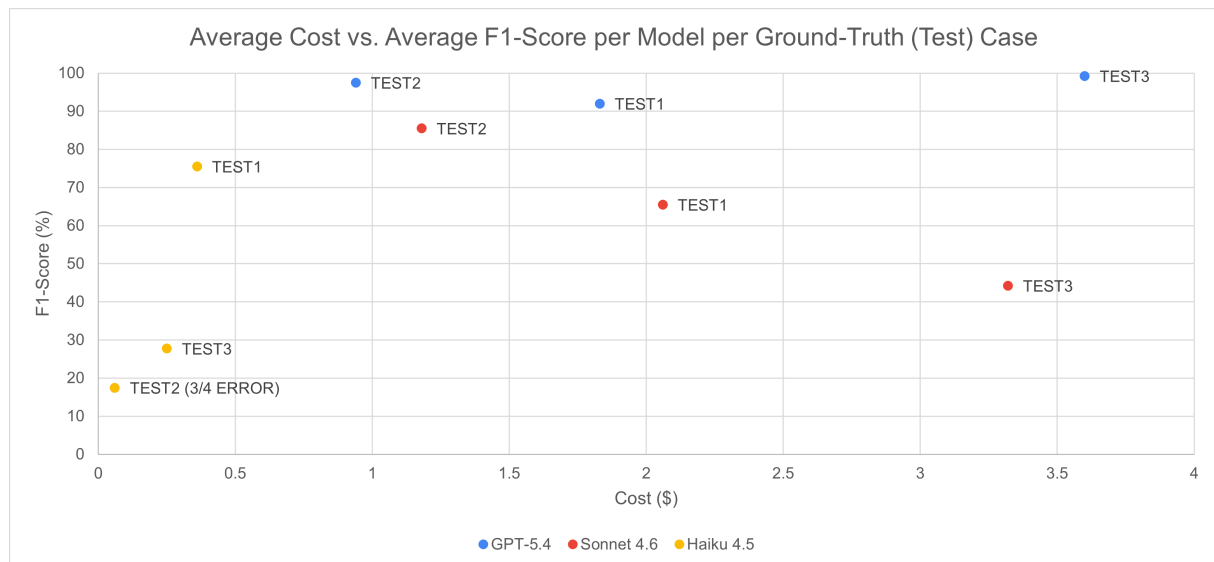


Figure 7.13: Average cost per case versus average F1-score per model and ground-truth scenario.

Each point represents the mean cost and mean F1-score across the four conversational archetypes for that scenario. In TEST1 and TEST2, Sonnet’s point sits slightly to the right of the corresponding GPT-5.4 point: marginally higher cost, substantially lower F1. In TEST3, this pattern reverses and Sonnet becomes the cheaper model (\$3.32 vs. \$3.60). That reversal is not incidental. Cost in this system scales directly with the number of Ontology Agents invoked, which in turn depends on how many entities the Orchestrator identifies. TEST3 presents seven trust entities. If Sonnet correctly identified all seven, its cost should exceed GPT-5.4 as it does in TEST1 and TEST2. The fact that it does not is cost-level evidence of entity omission. Sonnet’s Orchestrator dispatched fewer agents because it did not recognise all seven entities and the reduced agent count is what brings the cost below GPT-5.4. The recall collapse in TEST3 documented in Section 7.9.2 is therefore visible not only in the F1 data but in the cost structure of the run itself.

The cost-performance analysis reveals a result that contradicts the intuitive assumption that a more expensive model necessarily delivers better performance. Claude Sonnet 4.6 costs more than GPT-5.4 per case in TEST1 (\$2.06 vs. \$1.83) and TEST2 (\$1.18 vs. \$0.94), while delivering average F1-scores

of 65.5% vs. 92.0% in TEST1 and 85.5% vs. 97.5% in TEST2. Sonnet is both more expensive and less accurate than GPT-5.4 in the two most commercially representative onboarding scenarios.

Sonnet’s higher per-token pricing accounts for the cost premium in TEST1 and TEST2 despite similar or lower input token consumption than GPT-5.4. It costs more per run while delivering far fewer extracted fields.

Claude Haiku is four to fourteen times cheaper than GPT-5.4, with per-case costs of \$0.23–\$0.38. However, the three complete TEST2 failures mean that the effective cost-per-successful-case in the TEST2 scenario is not \$0.23 but a multiple thereof when the cost of failed runs and retry attempts is included. GPT-5.4 achieves the best quality-adjusted cost in the evaluation. It is the only model that clears the NFR-4 F1-threshold in all three test scenarios and it does so at a lower absolute cost than Sonnet in the TEST1 and TEST2 scenarios.

Figure 7.14 adds the latency dimension to the comparison.

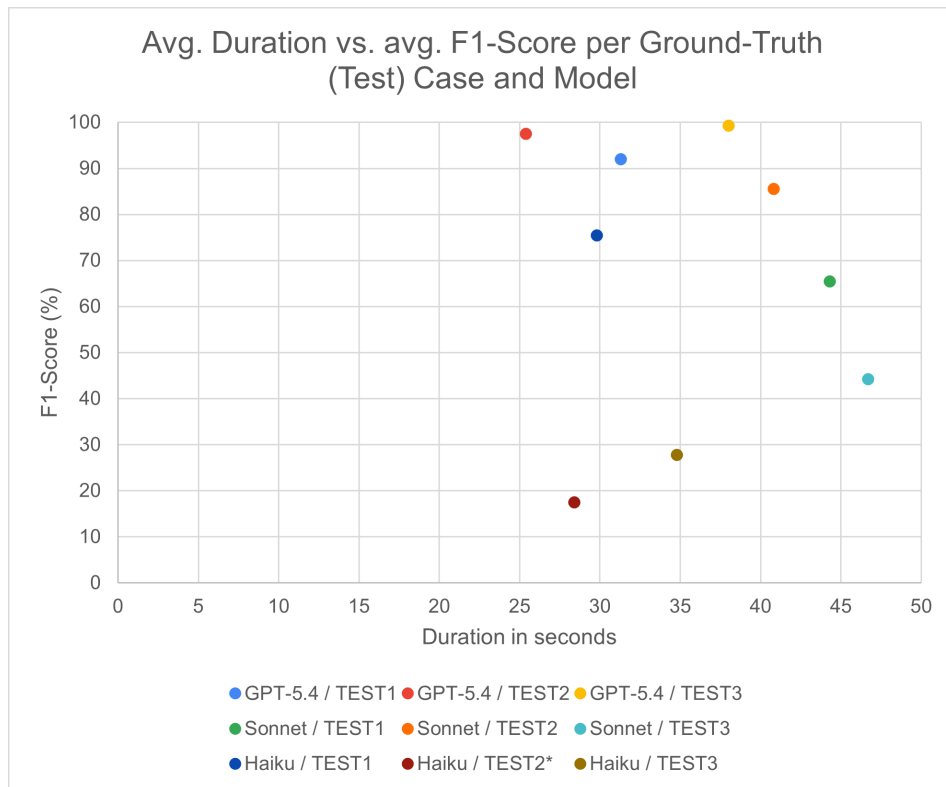


Figure 7.14: Average pipeline duration versus average F1-score per model and ground-truth scenario.

Sonnet is the slowest model across all scenarios. The exact contribution of each factor cannot be isolated from the recorded pipeline duration alone, but several are known to play a role. Sonnet accumulates more retry cycles than GPT-5.4 in most test cases, each adding a correction inference and an Atfinity round-trip. Beyond retries, LLM inference latency and the reasoning depth applied to complex structured extraction prompts both contribute to per-call duration. Anthropic models generally show higher reasoning latency on complex tasks, consistent with the duration gap visible here. Its three points range from 41 s (TEST2) to 47 s (TEST3) and sit to the right of every GPT-5.4 point on the duration axis. GPT-5.4 completes TEST2 in 25 s, TEST1 in 31 s and TEST3 in 38 s, faster than the corresponding Sonnet run in every case. The combined picture is straightforward: Sonnet takes longer and delivers lower F1 than GPT-5.4 in every scenario.

Haiku runs at a speed comparable to GPT-5.4: TEST1 at 30 s and TEST2 at 28 s. In TEST1, Haiku achieves 75.5% F1 at the same speed as GPT-5.4’s 92%. In TEST2 and TEST3, however, duration does not translate to quality: TEST2 is a pipeline failure (17.5% F1) and TEST3 achieves only 28% F1 at 35 s. Speed alone does not compensate for extraction failure.

Operational Metrics

Table 7.14 consolidates extraction quality, cost, token consumption and pipeline duration per scenario into a single reference for direct cross-model comparison.

GPT-5.4 achieves the highest F1 at comparable or lower cost than Sonnet in TEST1 and TEST2, while simultaneously running faster. In TEST1, GPT-5.4 delivers 92.0% F1 for \$1.83 in 31.3s, while Sonnet delivers 65.5% F1 for \$2.06 in 44.3s, more expensive, slower and less accurate. This three-way dominance holds across TEST1 and TEST2. In TEST3, GPT-5.4 costs \$3.60 and achieves 99.25% F1 in 38.0s; Sonnet costs \$3.32 and achieves only 44.25% F1 in 46.7s. As established in Section 7.9.2, the primary driver is entity omission. The Orchestrator fails to identify all seven trust entities, so entire persons are never submitted rather than individual fields being missed. Haiku costs far less per case but achieves only 27.75% F1 in TEST3 at \$0.25. In a compliance-driven banking context, an incomplete case submission requires manual Relationship Manager correction, which substantially reduces the nominal cost advantage.

Model	Scenario	Avg F1 (%)	Avg Cost (\$)	Avg Input Tok (K)	Avg Dur (s)
GPT-5.4	TEST1	92.0	1.83	712	31.3
GPT-5.4	TEST2	97.5	0.94	358	25.4
GPT-5.4	TEST3	99.25	3.60	1,415	38.0
Sonnet 4.6	TEST1	65.5	2.06	667	44.3
Sonnet 4.6	TEST2	85.5	1.18	378	40.8
Sonnet 4.6	TEST3	44.25	3.32	1,085	46.7
Haiku 4.5	TEST1	75.5	0.36	419	29.8
Haiku 4.5	TEST2*	17.5	0.06	66*	28.4*
Haiku 4.5	TEST3	27.75	0.25	294	34.8

Avg Input Tokens in thousands (K). *Haiku TEST2 reflects the email archetype only; three archetypes produced pipeline failures.

Table 7.14: Cross-model operational metrics per scenario.

Per-Scenario Token Breakdown

Table 7.15 provides a run-normalised per-scenario view: average input and output token consumption per single pipeline execution, enabling direct comparison between models at the scenario level.

Scenario	GPT-5.4 In	GPT-5.4 Out	Sonnet In	Sonnet Out	Haiku In	Haiku Out
TEST1	712,134	3,511	667,060	3,799	419,235	6,098
TEST2	357,685	2,715	378,339	2,932	265,678*	4,208*
TEST3	1,414,622	3,991	1,085,130	4,091	293,751	4,461

*Haiku TEST2 values reflect the email archetype only; three archetypes produced pipeline errors and logged no token metrics.

Table 7.15: Average input and output token consumption per scenario and model, per pipeline execution.

Figure 7.15 shows average input token consumption per model and scenario.

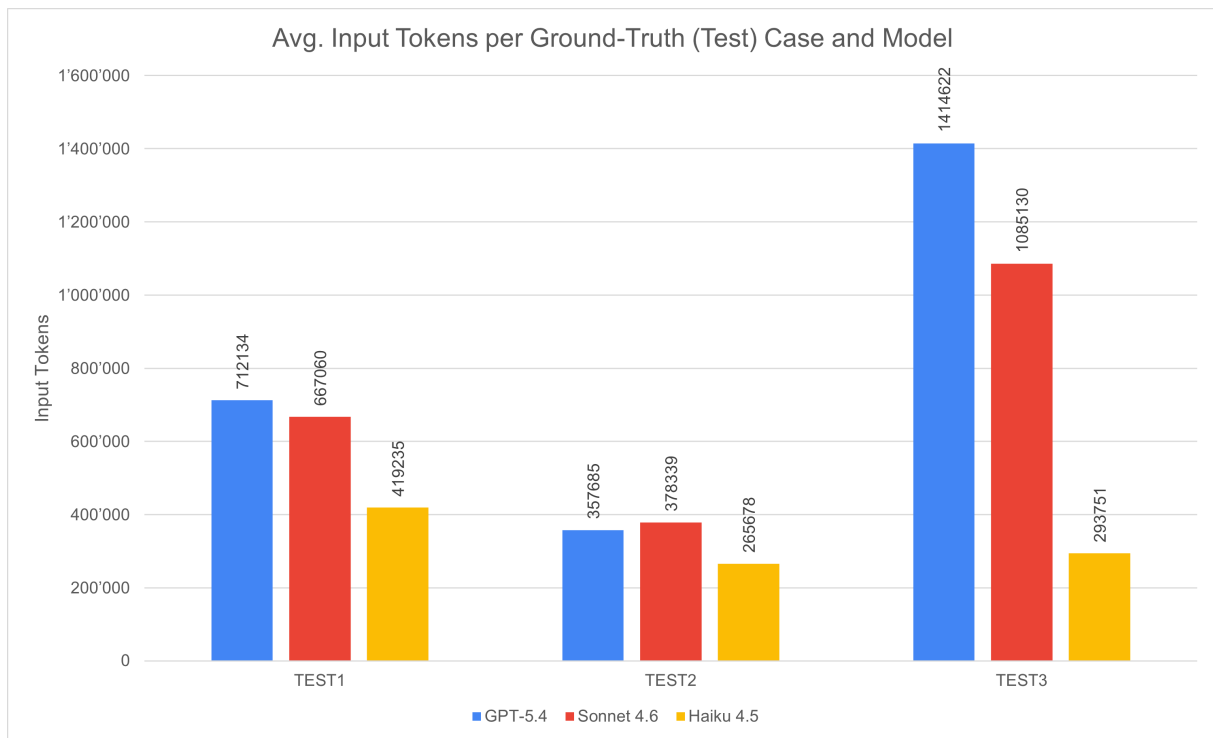


Figure 7.15: Average input token consumption per model and ground-truth scenario.

Input token volume roughly doubles with each step up in scenario complexity under GPT-5.4. From TEST2 (one entity) to TEST1 (three entities), tokens approximately double and from TEST1 to TEST3 (seven entities), they roughly double again. This pattern holds for GPT-5.4 across both steps, as visible in the chart. Claude Sonnet increases by roughly $1.8\times$ from TEST2 to TEST1, close to GPT-5.4 but below a clean doubling and its TEST3 figure (1,085,130 tokens) falls further below the expected second step, consistent with the entity omissions documented in Section 7.9.2, where fewer than seven agents were dispatched. Haiku makes the omission visible most starkly: its TEST3 total (293,751 tokens) is actually lower than TEST1 (419,235), despite TEST3 nominally involving more than twice the entities. Importantly, this token scaling does not translate into proportional latency. The parallel fan-out means TEST3 adds only $\approx 13s$ to TEST2's duration despite consuming nearly four times the tokens under GPT-5.4.

Haiku's input bars are consistently the shortest across all scenarios: 41% fewer tokens than GPT-5.4 in TEST1 and 79% fewer in TEST3 despite receiving identical schema inputs. Sonnet shows the same pattern at a smaller magnitude: 6% fewer than GPT-5.4 in TEST1 and 23% fewer in TEST3. These gaps are tokenizer artefacts. The Anthropic tokenizer encodes the Bank of Luxembourg's ontology schemas more compactly than the OpenAI tokenizer. They do not reflect differences in schema coverage. In TEST3, however, a separate effect compounds the gap for Sonnet. The entity omission documented in Section 7.9.2 means fewer agents were dispatched and therefore fewer schema chunks were sent, reducing input tokens beyond what the tokenizer difference alone would explain.

Figure 7.16 shows average output token consumption per model and scenario.

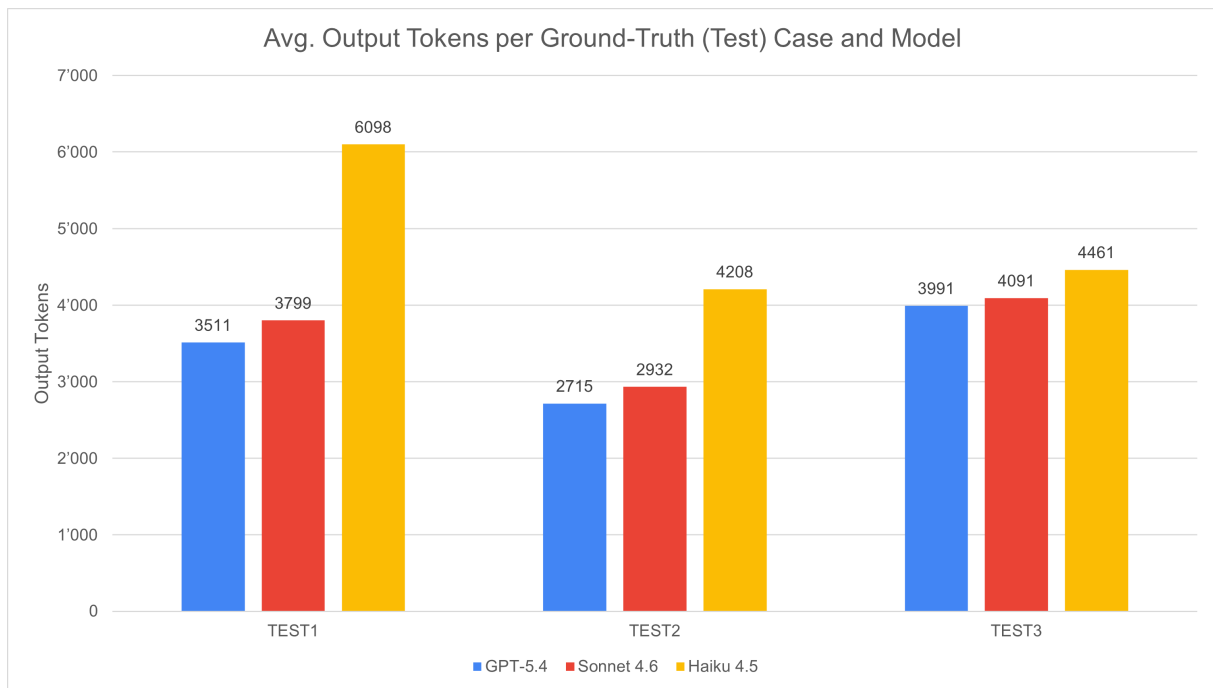


Figure 7.16: Average output token consumption per model and ground-truth scenario.

Output tokens do not scale with input tokens in the same way. GPT-5.4 outputs 2,715 tokens in TEST2 and 3,991 in TEST3, an increase of 47% despite a $3.95\times$ increase in input. The output is bounded by the schema. Each Ontology Agent response fills a fixed set of fields and a larger input context does not produce a proportionally larger response.

Haiku is the clear outlier. It produces 6,098 output tokens in TEST1, 74% more than GPT-5.4's 3,511 for the same scenario. Two factors contribute to this. Each retry cycle produces an additional Ontology Agent correction response, adding output tokens per run. Beyond retries, Haiku also populates more fields per schema call. It submits values where Sonnet leaves fields empty, resulting in denser responses. Yet Haiku's TEST1 F1 of 75.5% is below GPT-5.4's 92.0%. More output tokens do not indicate better extraction quality. Sonnet outputs 3,799 tokens in TEST1, almost identical to GPT-5.4, yet achieves 26.5 percentage points less F1. Output volume is not a proxy for extraction accuracy in either direction. Figure 7.17 combines input volume, output volume and cost into a single view.

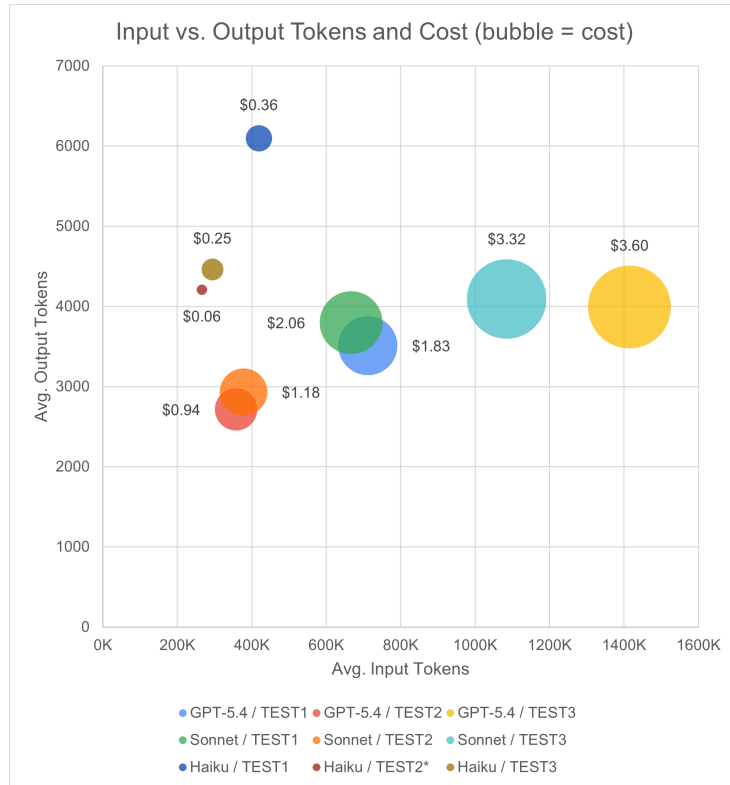


Figure 7.17: Average input tokens versus average output tokens per model and ground-truth scenario, with bubble area proportional to cost.

Input token volume scales directly with entity count. TEST2, with one entity, sits in the left cluster. TEST1, with three entities, sits in the middle. TEST3, with seven entities, sits on the far right. Each additional entity means one additional Ontology Agent invocation and one additional schema-filling API call, scaling the input token count linearly with entity count. Higher input volume therefore drives higher cost and this is visible in the bubble areas growing from left to right.

Output tokens remain largely stable regardless of input volume. Most points cluster between approximately 2,800 and 4,500 output tokens across all models and scenarios. The output is bounded by the schema. Each Ontology Agent response fills a fixed set of fields and that response size does not scale with how large the input context was.

Haiku TEST1 is the clear outlier at approximately 6,100 output tokens, the highest in the chart. Two factors contribute to this. First, the retry correction cycles each produce an additional Ontology Agent response, since every Atfinity rejection triggers a correction output, adding to the total output token count for that run. Second, Haiku populates more fields per schema call than Sonnet, producing denser extraction responses where Sonnet would leave fields empty.

The Sonnet TEST3 bubble (\$3.32) is smaller than the GPT-5.4 TEST3 bubble (\$3.60) and sits to its left on the input axis. This is the same entity omission signal described in Section 7.10.3. In TEST1 and TEST2, Sonnet's cost exceeds GPT-5.4's despite similar or lower input token consumption, which reflects its higher per-token pricing. In TEST3 this reverses: Sonnet's input token count falls to 1,085,130 while GPT-5.4's remains at 1,414,622. That drop can only happen if fewer agents were invoked. Fewer agents means fewer entities were identified by the Orchestrator. The cost, the bubble size and the X-axis position all independently confirm the same omission.

7.11 Limitations

The conducted evaluation is subject to the following limitations:

- The evaluation suite comprises twelve test cases derived from three ground-truth cases. While these cases cover a representative range of case types, joint individual accounts, high-density single-party accounts and complex trust structures, they do not exhaustively cover all onboarding scenarios available in the OST Atfinity deployment. Edge cases involving unusual ontology combinations, non-standard role assignments or adversarial input are not represented.
- The evaluation was not conducted against adversarial or intentionally malformed input. All test prompts were generated by a controlled synthetic data generator at $T = 0.7$ and represent plausible Relationship Manager inputs rather than worst-case conditions.
- Mixed-model configurations, assigning different LLMs to different agent roles, were not empirically evaluated. The analysis in Section 7.7.3 is therefore a theoretical inference from single-model results. A dedicated mixed-model evaluation study would be required to validate the cost-performance trade-off claims.
- Test data is synthetic and may not fully capture the vocabulary, phrasing patterns and implicit domain knowledge of experienced Relationship Managers in a live production environment.
- GPT-5.4-mini produced no valid evaluation results due to systematic Atfinity payload rejection. No partial metric data is available for this model, preventing quantitative comparison at any F1 level.
- No per-model prompt optimisation was performed; the same system prompts were used for every agent across all four model configurations, as noted in Section 7.8. Different providers can respond differently to identical instruction phrasing, so model-specific prompt tuning, particularly for the lower-tier models such as Claude Haiku 4.5 and GPT-5.4-mini, could yield improved results and represents a possible direction for future work.
- The OST VM hardware (2 vCPU, 4096 MB RAM) represents a resource-constrained deployment environment. Latency results may differ on production infrastructure with higher CPU core counts and memory, particularly for TEST3 test cases that invoke seven simultaneous Ontology Agents.
- Open-source models were not included in the evaluation. The decision was made after GPT-5.4-mini, the smallest paid model evaluated, produced no valid results due to systematic payload rejection, indicating that models at the lower end of the capability spectrum are not viable for this task in its current form. Given this finding, extending the evaluation to open-source models of comparable or smaller scale was not expected to yield meaningful positive results and was excluded on those grounds as well as time constraints.
- The evaluation framework measures extraction accuracy exclusively at the level of the complete end-to-end pipeline. Individual agent performance was not evaluated in isolation. The F1, precision and recall metrics reported in this chapter attribute extraction quality to the composed output of the full multi-agent pipeline rather than to any single agent independently. Each sub-agent in the system exposes a direct HTTP endpoint that is callable outside the full pipeline context. The Ontology Agents accept calls to `POST /process` with a plain-text prompt and return a field-extraction JSON for their assigned ontology directly; the Process Agent accepts calls to `POST /determine_process`; and the Prompt Organizer Agent accepts calls to `POST /organize`. Calling these endpoints with controlled test inputs and comparing their outputs against ontology-specific reference data would allow precision and recall to be attributed to specific agent roles and ontology types rather than to the pipeline as a whole. Isolated agent evaluation was outside the scope of the current thesis and is identified as a methodological extension for future work.

7.12 Delivered Artefacts

The following artefacts were produced as part of the project:

- The Multi-agent Preemptive Single-Shot architecture as the primary end product, comprising all agent containers, the schema synchronisation pipeline, the CI/CD deployment infrastructure and the LibreChat-based frontend integration.
- A deterministic evaluation framework comprising a synthetic test-case generator, an end-to-end evaluation runner and a canonical test suite, enabling full reproducibility of the evaluation results presented in this chapter.
- This bachelor thesis as the technical documentation artefact, providing the architectural rationale, implementation detail and evaluation analysis required to understand, reproduce and extend the system.

Part V

Project Documentation

Chapter 8

Project- and Time Management

This chapter describes the project management approach applied throughout the Bachelor Thesis. It covers the chosen development methodology, team composition, responsibilities, communication practices, planning tools and a structured risk analysis conducted and updated at each sprint iteration.

8.1 Project Planning

The following sections describe how the project was managed and planned, from methodology selection and team organisation through long-term planning, milestone definition and iterative sprint execution.

8.1.1 Methodology

The Bachelor Thesis was conducted using Scrum+, an OST-developed hybrid methodology that combines the structural discipline of the Rational Unified Process (RUP) with the iterative flexibility of Scrum. At the macro level, the project is organised into four RUP phases: Inception, Elaboration, Construction and Transition. Each phase defines a set of milestone-oriented goals and expected deliverables, providing a long-term roadmap for the project. At the micro level, development is executed within Scrum-style two-week iterations. Each iteration includes a sprint planning session, active development work, a sprint review with all stakeholders and a retrospective. This dual structure allows the project to maintain strategic coherence across phases while adapting to new findings and changing requirements within each sprint. Scrum+ is particularly well-suited to a research-oriented thesis environment, where architectural decisions are exploratory in nature and evaluation results must continuously inform the development direction.

8.1.2 Project Members

Table 8.1 lists the project team members, their roles and affiliations.

Role	Name	Comment
Client / Partner	Atfinity (Thorben Croisé)	Industrial Partner of OST and CTO of Atfinity
Supervisor	Prof. Dr. Mitra Purandare	
Co-Examiner	Dr. Marc Rösli	
Developer, BA Author	Arnel Veladzic	
Developer, BA Author	Fabio Gomes Silva	

Table 8.1: Project roles and team members

8.1.3 Responsibilities

Each team member is not solely responsible for executing all tasks within their assigned area. Instead, they oversee their respective components, ensuring alignment and progress. Arnel Veladzic is responsible for scheduling and managing meetings, ensuring the project remains on track. Fabio Gomes Silva also contributes to the planning process and actively commits to the scheduled milestones.

Arnel Veladzic:

- Project planning
- Product Architecture and Implementation
- Quality Measures: Implementation and Evaluation (Assistance)
- Documentation

Fabio Gomes Silva:

- Test Concept
- Quality Measures: Implementation and Evaluation
- Product Architecture and Implementation (Assistance)
- Documentation

8.1.4 Collaboration and Meetings

Since the project is conducted using an agile methodology, most meetings are aligned with Scrum ceremonies.

- Sprint Review/Retro: Every second Sunday 10:00 – 11:00
- Sprint Planning: Every second Sunday 11:00 – 12:00
- Weekly Sync with Supervisor and Atfinity: Every Thursday 15:00 – 15:30
- Additional Meetings: Scheduled when needed

8.1.5 Planning Tools

For effective tracking and management, the following tools are used:

- Jira: Task tracking and sprint backlog organisation
- Confluence: Collaborative knowledge sharing and notes
- Miro: Visual project planning
- Clockify Plugin for Jira: Time tracking

8.1.6 Long-Term Planning

The project follows the Scrum+ structure, combining high-level phase planning with iterative sprint execution. The overall project plan spans approximately 15 to 16 weeks, structured into 8.5 two-week iterations. The final half-iteration covers only the submission period and concludes on 12.06.2026. The plan is organised according to the four RUP phases: Inception, addressing project setup and initial research; Elaboration, focused on architectural exploration and stakeholder alignment; Construction, comprising the core implementation and evaluation work; and Transition, covering finalization and documentation. As illustrated in Figure 8.1, milestones are placed at key phase boundaries and serve as formal checkpoints for evaluating the completeness of each development stage. Concrete work items are not pre-defined for all iterations; instead, they are identified through sprint planning at the start of each iteration, in accordance with the Scrum+ methodology.

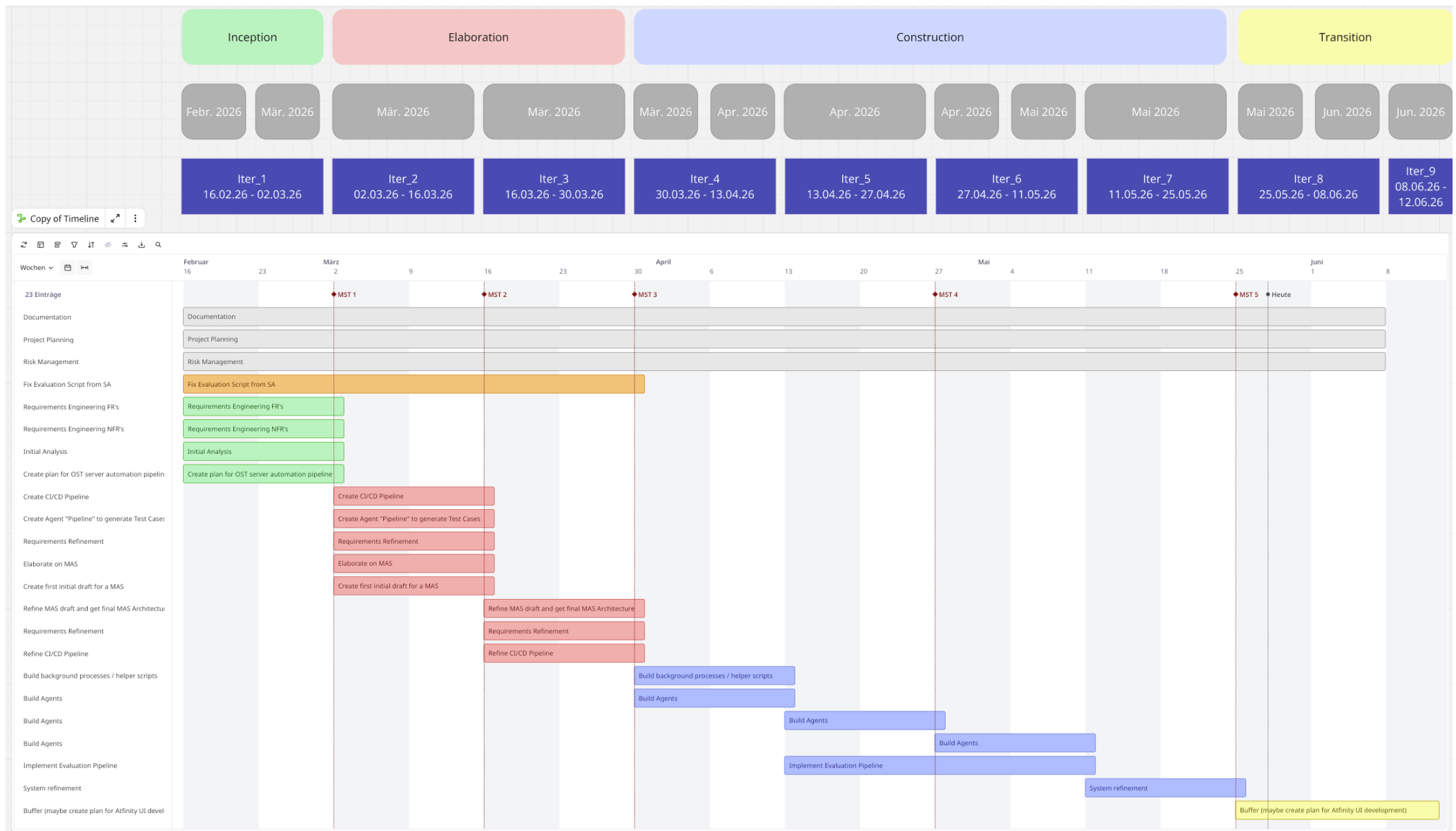


Figure 8.1: Final Project Plan

8.1.7 Milestones

The project is structured around five clearly defined milestones to indicate when key requirements and deliverables are met. Each milestone is associated with a Definition of Done, which specifies the criteria that must be fulfilled for the milestone to be considered complete. Concrete tasks within each milestone are identified during sprint planning at the beginning of each iteration.

Milestone 1: Project Initiation and Foundation

This milestone marks the completion of the project setup phase. It covers the evaluation of issues identified in the Study Thesis, the configuration of the development environment for the Bachelor Thesis scope and the formulation of an initial research direction.

Definition of Done:

- Documentation structure established for the BA scope
- Known Study Thesis issues evaluated and addressed
- Functional and non-functional requirements documented
- Initial project plan created and reviewed with supervisor
- Key risks identified and captured in the risk register
- Development environment fully operational

Milestone 2: Architectural Exploration

This milestone marks the completion of the architectural exploration phase. Multiple candidate multi-agent system architectures are drafted and assessed and a stakeholder review with Atfinity and the supervisor validates the feasibility of the proposed approaches.

Definition of Done:

- Documentation updated to reflect BA scope and research context
- The three architecture variants from the preceding study thesis were re-evaluated in the context of this bachelor thesis
- Stakeholder review conducted and feedback incorporated
- Technology stack validated through prototype experiments
- Preliminary deployment roadmap established

Milestone 3: Architecture Finalisation

This milestone marks the selection and approval of the final multi-agent system architecture. The selected design serves as the definitive blueprint for implementation in the subsequent construction phase.

Definition of Done:

- Final multi-agent architecture approved by supervisor and industrial partner
- Architecture decision rationale documented for each major design choice
- Deployment roadmap finalised and aligned with the sprint plan
- All prior architectural variants documented for comparative evaluation

Milestone 4: Core System Deployment

This milestone marks the successful deployment of a functional end-to-end system. The deployed MVP demonstrates that the multi-agent pipeline can receive a natural-language onboarding prompt and produce a valid, structured JSON payload for submission to Atfinity's CMS.

Definition of Done:

- Multi-agent system deployed and accessible via the MCP interface
- End-to-end onboarding scenario tested successfully
- Automated evaluation pipeline is in development
- System behaviour manually validated against at least one ground truth case

Milestone 5: Final Automation and Delivery

This milestone marks the conclusion of the project. The multi-agent system is fully implemented and operationally stable, the automated evaluation pipeline is integrated into the CI/CD infrastructure and all documentation meets the Bachelor Thesis submission standard.

Definition of Done:

- All research questions answered and documented
- Automated evaluation integrated into the GitLab CI/CD pipeline
- Final system evaluation completed across all 3 ground truths
- Complete Bachelor Thesis documentation finalised
- Code repository clean, documented and tagged for submission

8.1.8 Short-Term Planning

Since the project is conducted using Scrum+, work is organised into two-week iterations. A detailed short-term plan is not prescribed at the start of the project. Instead, concrete tasks are identified during sprint planning at the beginning of each iteration, based on the current state of the backlog and the goals defined for the active phase.

During sprint planning, upcoming work items are reviewed and prioritised, larger requirements are broken down into actionable tasks and effort estimates are discussed collaboratively. Tasks are assigned based on current priorities and individual capacity.

This iterative approach ensures that the project remains responsive to new technical findings and evaluation results, while maintaining steady progress toward the defined milestones.

8.2 Time Management

The working hours of the team are systematically tracked throughout the project using clockify. Time spent is recorded at the project level rather than being assigned to individual tasks or issues. Each team member is responsible for logging their working time accurately and consistently to ensure transparency and reliable effort tracking across the project.

8.2.1 Time Management Statistics

Figure 8.2 compares actual hours per sprint against the 96-hour two-week target. Figure 8.3 shows the effort split: Arnel Veladzic contributed 469.88 hours and Fabio Gomes Silva 399 hours, totalling 868.88 hours against a target of 806 hours.

Three sprints account for the bulk of the overrun. Sprint 2 saw the steepest spike, driven by first contact with the Atfinity tenant and the architectural exploration phase where the learning curve was highest. Sprint 5 produced the second-largest overrun during full multi-agent pipeline integration and the first end-to-end evaluation runs. Sprint 7 recorded the highest effort, coinciding with the convergence of final evaluation implementation, results analysis and the primary documentation phase. Sprint 9 covers only a single submission week and is below target as expected.

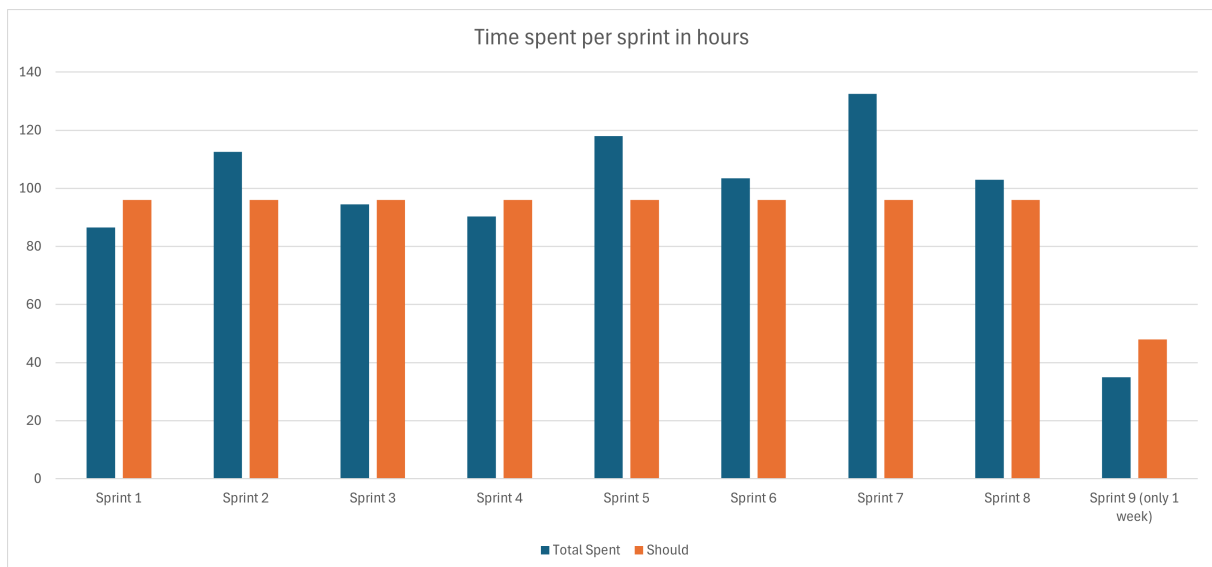


Figure 8.2: Time spent per sprint of the whole team

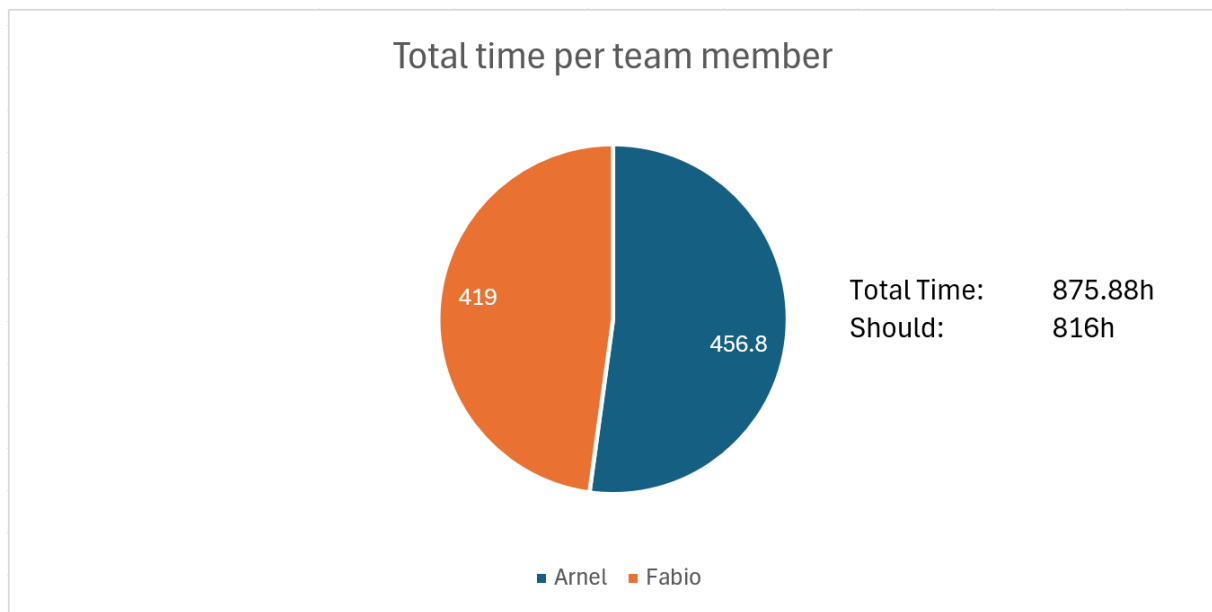


Figure 8.3: Time spent per team member for whole project

8.3 Risk Management

To manage project risks effectively, a structured risk analysis is conducted at the start of the project and updated at each sprint iteration. The risk register is reviewed during sprint retrospectives to reflect changes in probability and impact based on the current state of the project. Figures 8.4 through 8.11 show the risk matrix as recorded at the end of each sprint; the subsections below describe the specific risk changes per sprint.

8.3.1 Risks and Mitigations

Table 8.2 presents the identified risks for the Bachelor Thesis, together with their descriptions, mitigation strategies and responsible owners. Blue rows represent project planning risks; green rows represent technical risks.

Risk	Description	Mitigation	Owner
Not Enough Buffer Planned	Insufficient time reserves for unexpected delays	Add time buffers to milestones; adaptive sprint planning	Arnel Veladzig and (Project Team)
Scope Creep	Uncontrolled expansion of requirements or features	Strict backlog prioritisation; enforced sprint goals	Arnel Veladzig and (Project Team)
Underestimation of Man-Hours	Effort estimates are overly optimistic	Regular re-estimation per sprint; retrospective calibration	Arnel Veladzig and (Project Team)
Unclear / Changing Requirements	Requirements are incomplete or subject to late changes	Continuous stakeholder alignment and clarification sessions	Project Team
Technology Dependency	Reliance on external frameworks, APIs or tools	Early prototype validation; identify fallback alternatives	Project Team
Bugs and Failures	Software defects affect system functionality or stability	Continuous testing, incremental integration, CI/CD pipeline	Project Team
Misconfig of Architecture	Incorrect system design or configuration decisions	Architecture reviews and stakeholder validation sessions	Project Team
Agent Coordination Failure	Communication breakdown between agents due to protocol errors or LLM tool-call failures	Structured inter-agent message schemas; retry logic; agent-level health monitoring	Project Team
Evaluation Bias / GT Quality	Ground truth data is incomplete or biased, distorting evaluation metrics and architectural comparisons	Multi-case ground truth validation; deterministic field-level evaluation engine	Project Team
LLM Hallucination	LLM generates plausible but incorrect field values, causing silent data corruption in the CMS	Precision/recall/F1 evaluation; Atfinity CMS field validation as secondary detection layer	Project Team
LLM API Cost Overrun	High token consumption from multi-agent processing or stochastic test repetition exceeds budget	Per-model token and cost tracking; configurable run counts via <code>--runs</code> flag	Project Team

Table 8.2: Risks and their mitigations (colour matching to risk matrix)

8.3.2 Risk Matrix in Sprint 1

To support effective risk management, a structured risk analysis is conducted to identify the key risks associated with the project. These risks are reviewed and updated at each sprint iteration to reflect the current project state.

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks

Risks related to individual team members were deliberately excluded from the analysis, as the project is carried out by a team of only two developers. In this context, a detailed assessment of personnel-related risks was considered neither meaningful nor proportionate.



Figure 8.4: Sprint 1; Risk Matrix

Misconfiguration of Architecture

This risk is positioned at likely probability and extreme impact. At the start of the project, the architecture of the multi-agent system has not yet been designed. Incorrect foundational decisions made during this phase would propagate through all subsequent implementation work and require extensive rework to correct.

Underestimation of Man-Hours

This risk is positioned at likely probability and extreme impact. The problem domain, including multi-agent coordination, MCP protocol behaviour and Atfinity's CMS field semantics, is largely unfamiliar at the outset. Effort estimation under these conditions is inherently uncertain and the small team size amplifies the consequences of any shortfall.

Bugs and Failures

This risk is placed at unlikely probability and major impact. While early sprints focus primarily on research and design rather than implementation, the introduction of any prototype code carries the risk of defects. The impact is assessed as major, since architectural bugs in foundational components would be difficult to detect and correct later.

Agent Coordination Failure

This risk is positioned at unlikely probability and extreme impact. The multi-agent architecture requires reliable communication between independently running agent services. Any failure in the coordination protocol, whether due to schema mismatches, LLM tool-call errors or network issues, would prevent the system from functioning as a whole.

Scope Creep

This risk is assessed at likely probability and moderate impact. In Sprint 1, the project scope is still being defined and the breadth of the problem domain increases the likelihood of scope expansion. The impact is moderated by the fact that requirements are still largely conceptual at this stage.

Evaluation Bias and Ground Truth Quality

This risk is placed at unlikely probability and major impact. The quality of the automated evaluation depends entirely on the integrity of the ground truth data. Incomplete or unrepresentative ground truth cases would produce misleading precision and recall metrics, undermining the validity of all subsequent architectural comparisons.

Unclear or Changing Requirements

This risk is positioned at moderate probability and moderate impact. In Sprint 1, requirements have not yet been fully stabilized. The combination of stakeholder expectations, academic research goals and emerging technical constraints creates a significant source of uncertainty.

Technology Dependency

This risk is assessed at moderate probability and extreme impact. The system relies on the MCP protocol, Atfinity's proprietary CMS API and multiple LLM providers. Unexpected breaking changes or limitations in any of these external dependencies could significantly affect the feasibility of the planned architecture.

Not Enough Buffer Planned

This risk is placed at moderate probability and moderate impact. The project timeline must accommodate both development and academic documentation deliverables within a fixed 15 to 16-week window. Underestimating the effort required for either area could result in insufficient time reserves during critical late-project phases.

LLM Hallucination

This risk is positioned at rare probability and extreme impact. The likelihood of observable hallucination is low in Sprint 1, as the evaluation framework has not yet been deployed. However, the potential for the system to generate plausible but structurally invalid or semantically incorrect field values without detection is registered early due to the severity of the downstream consequences in the CMS.

LLM API Cost Overrun

This risk is placed at unlikely probability and minor impact. API costs are minimal during the early design-focused sprints. The risk is registered to ensure that later sprints involving stochastic test repetition and multi-model comparison remain within budget.

8.3.3 Risk Matrix in Sprint 2

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.5: Sprint 2; Risk Matrix

Misconfiguration of Architecture

The risk remains at likely probability and extreme impact. Architectural decisions are being actively made in this sprint and the absence of a validated design means that fundamental configuration errors are still highly probable and would have far-reaching consequences.

Underestimation of Man-Hours

The risk moved to moderate probability and extreme impact. Initial sprint experience provided a first calibration point for effort estimation, reducing the likelihood of severe underestimation. The impact remains extreme due to the small team size and the fixed academic deadline.

Bugs and Failures

This risk remains at rare probability and major impact. Development activity in Sprint 2 is still primarily design-focused. The probability of defects is low, but bugs introduced in foundational components would carry major impact if not caught early.

Agent Coordination Failure

This risk remains at rare probability and extreme impact. No inter-agent communication has been implemented yet. The risk is tracked at this stage to ensure that coordination protocols are given adequate consideration during architectural design.

Scope Creep

This risk remains at likely probability and moderate impact. Architectural exploration in Sprint 2 involves examining multiple design alternatives, which increases the likelihood of expanding scope. Strict sprint goal enforcement mitigates the impact.

Evaluation Bias and Ground Truth Quality

This risk remains at unlikely probability and moderate impact. The evaluation framework is not yet implemented. However, design decisions made now regarding test case structure and ground truth representation will influence evaluation validity in later sprints.

Unclear or Changing Requirements

This risk remains at moderate probability and moderate impact. Stakeholder feedback from the first sprint review may refine or extend requirements. Regular clarification sessions reduce the likelihood of misalignment.

Technology Dependency

The risk escalated to likely probability and extreme impact. Sprint 2 involves first contact with the Atfinity CMS API and MCP tooling. The probability of encountering unexpected constraints or API limitations is high and such issues could force architectural redesigns.

Not Enough Buffer Planned

This risk remains at moderate probability and moderate impact. The available buffer has not yet been consumed, but the complexity of the architectural exploration phase makes adequate buffer planning a continued concern.

LLM Hallucination

This risk remains at rare probability and extreme impact. The low probability reflects the absence of live LLM integration at this stage. The extreme impact classification is retained as a forward-looking assessment, given the potential for silent data corruption once the system becomes operational.

LLM API Cost Overrun

This risk remains at unlikely probability and minor impact. API calls in Sprint 2 are limited to prototype experiments. Cost exposure is minimal.

8.3.4 Risk Matrix in Sprint 3

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.6: Sprint 3; Risk Matrix

Misconfiguration of Architecture

The risk moved to moderate probability and extreme impact. As concrete architecture components are being implemented, the probability of misconfiguration decreases compared to the design-only phase. The extreme impact is retained because fundamental architectural errors at this stage would require substantial rework.

Underestimation of Man-Hours

This risk remains at moderate probability and extreme impact. Sprint 3 introduces the first substantial implementation work. Experience from earlier sprints improved estimation, but the complexity of multi-agent orchestration continues to introduce uncertainty.

Bugs and Failures

This risk remains at unlikely probability and major impact. With implementation beginning, the likelihood of defects increases relative to earlier sprints, though it remains unlikely. Bugs in foundational components such as agent routing or schema validation would carry major impact.

Agent Coordination Failure

The risk escalated to moderate probability and major impact. Inter-agent communication protocols are being implemented in this sprint. Configuration errors or protocol mismatches in the early coordination layer have a moderate probability of occurring and would directly impede end-to-end system integration.

Scope Creep

This risk decreased to moderate probability and moderate impact. Sprint goals are better defined and stakeholder feedback has stabilized requirements. The risk remains moderate due to the exploratory nature of multi-agent architecture.

Evaluation Bias and Ground Truth Quality

This risk remains at unlikely probability and moderate impact. Ground truth data preparation is beginning alongside implementation. The risk is managed through careful selection of representative onboarding cases spanning diverse ontological structures.

Unclear or Changing Requirements

The probability decreased to unlikely with major impact. Continued stakeholder alignment has clarified the majority of requirements, though changes prompted by implementation findings remain possible and would still have significant consequences.

Technology Dependency

The risk decreased to unlikely probability and extreme impact. Core technology choices have been validated through Sprint 2 prototyping. The extreme impact is retained, as any blocking issue with the Atfinity API or MCP tooling would still require architectural adjustment.

Not Enough Buffer Planned

This risk decreased to unlikely probability and moderate impact. Buffer usage in the first two sprints was within acceptable bounds, reducing the likelihood of time pressure. The moderate impact reflects the risk of delays during the complex implementation phase.

LLM Hallucination

The risk escalated to moderate probability and extreme impact. With LLM integration now active, the system is capable of generating field values. The moderate probability reflects that hallucinations are known to occur with inference-based models under complex structured output constraints and the evaluation framework is not yet fully operational to detect them systematically.

LLM API Cost Overrun

This risk remained at rare probability and minor impact. Token consumption is still limited to targeted prototype calls and costs remain well within acceptable bounds.

8.3.5 Risk Matrix in Sprint 4

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.7: Sprint 4; Risk Matrix

Misconfiguration of Architecture

The risk decreased to moderate probability and major impact. Architectural validation during Sprint 3 reduced the likelihood of fundamental misconfiguration. The impact decreased from extreme to major, reflecting that the core design is now partially validated through integration testing.

Underestimation of Man-Hours

The risk decreased to between unlikely and moderate probability, with major impact. Improved task breakdown and experience from previous sprints reduced estimation uncertainty. The major impact reflects the continued consequence of effort shortfalls on milestone delivery.

Bugs and Failures

The risk escalated to moderate probability and extreme impact. Increased system complexity and the integration of multiple agent components raise the likelihood of defects. Automated testing through the CI/CD pipeline acts as the primary mitigation measure.

Agent Coordination Failure

The risk escalated to moderate probability and extreme impact. Full inter-agent coordination is now active and the probability of protocol-level failures during integration testing is moderate. Any coordination breakdown at this stage would directly block end-to-end functionality.

Scope Creep

The risk remains at moderate probability and moderate impact. The MVP target creates pressure to add features, but clearly defined sprint goals and strict backlog prioritization contain the scope.

Evaluation Bias and Ground Truth Quality

The risk remains at unlikely probability and moderate impact. Ground truth preparation is ongoing. Expanding the test suite to cover additional CMS ontology configurations reduces bias but adds validation effort.

Unclear or Changing Requirements

The risk decreased to rare probability and minor impact. Requirements are now substantially stable and the system is being tested against a fixed set of ground truth cases.

Technology Dependency

The risk decreased to moderate probability and moderate impact. Core technologies have been validated, but the integration of additional ontology types continues to expose edge cases in the Atfinity API.

Not Enough Buffer Planned

This risk remains at unlikely probability and moderate impact. Buffer planning has been continuously reviewed and sprint velocity has been consistent with initial estimates.

LLM Hallucination

The risk remains at moderate probability and extreme impact. The system is fully operational and generating CMS field values in each test run. The deterministic evaluation engine now provides a mechanism for detecting hallucinations as field-level mismatches in the precision and recall metrics.

LLM API Cost Overrun

The risk decreased to rare probability and minor impact. Token consumption is tracked per model and per test run and costs remain manageable within the current testing volume.

8.3.6 Risk Matrix in Sprint 5

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.8: Sprint 5; Risk Matrix

Misconfiguration of Architecture

The risk decreased to unlikely probability and moderate impact. The core architecture has been validated through end-to-end testing. Remaining misconfiguration risks are isolated to specific configuration parameters rather than fundamental design decisions.

Underestimation of Man-Hours

The risk decreased to unlikely probability and minor impact. Consistent sprint delivery over four iterations has established a reliable velocity baseline. Remaining effort is predominantly implementation and documentation work with well-understood scope.

Bugs and Failures

The risk escalated to likely probability and extreme impact. Sprint 5 represents peak system complexity, with all agent components active simultaneously. The high probability of defects during this integration phase makes thorough automated testing and continuous monitoring critical mitigation measures.

Agent Coordination Failure

The risk escalated to likely probability and extreme impact. Full multi-agent pipeline execution is now the primary development activity. Coordination failures involving message routing errors, LLM tool-call timeouts or agent state inconsistencies are likely to occur during intensive end-to-end testing.

Scope Creep

The risk decreased to rare probability and moderate impact. Core functionality is substantially complete and the remaining scope is well-defined. The moderate impact is retained as a residual caution.

Evaluation Bias and Ground Truth Quality

The risk increased to moderate probability and moderate impact. Evaluation results are now actively used to drive architectural decisions. Any bias in the ground truth data or inconsistency in the deterministic evaluation engine would distort the comparative analysis between system variants.

Unclear or Changing Requirements

The risk remains at rare probability and minor impact. Requirements are stable and validated against the ground truth dataset.

Technology Dependency

The risk decreased to unlikely probability and moderate impact. All major technology dependencies have been resolved through prior implementation sprints.

Not Enough Buffer Planned

The risk remains at unlikely probability and minor impact. Sprint velocity is stable and remaining milestones are clearly scoped.

LLM Hallucination

The risk remains at moderate probability and extreme impact. Stochastic repetition via the `--runs` flag increases statistical exposure to hallucinations across multiple inference calls. The deterministic evaluator captures these as false positive or false negative events in the precision and recall metrics.

LLM API Cost Overrun

The risk increased to moderate probability and moderate impact. Stochastic test repetition and multi-model comparative evaluation significantly increase token consumption. Cost tracking via the per-model `COST_PER_1M_TOKENS` table provides visibility, but active budget discipline becomes important in this sprint.

8.3.7 Risk Matrix in Sprint 6

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.9: Sprint 6; Risk Matrix

Misconfiguration of Architecture

The risk decreased to rare probability and moderate impact. The architecture is finalised and all major configuration decisions have been validated through repeated end-to-end test runs.

Underestimation of Man-Hours

The risk remains at unlikely probability and minor impact. Remaining work is primarily documentation and final evaluation, with well-understood effort requirements.

Bugs and Failures

The risk remains at likely probability and extreme impact. The full multi-agent pipeline is under continuous load from evaluation runs. System-level defects and edge-case failures remain likely given the variety of input prompts exercised by the 12-case test suite.

Agent Coordination Failure

The risk decreased to moderate probability and extreme impact. Inter-agent communication is stable under normal conditions, but the probability of coordination failure under edge-case prompts or LLM timeout scenarios remains moderate.

Scope Creep

The risk decreased to rare probability and moderate impact. All requirements are met; residual scope expansion is unlikely.

Evaluation Bias and Ground Truth Quality

The risk increased to moderate probability and major impact. As evaluation data begins to be analysed for the thesis discussion, the validity of the ground truth becomes increasingly critical. Any previously undetected bias in the test cases would affect the academic conclusions drawn from the results.

Unclear or Changing Requirements

The risk remains at rare probability and minor impact.

Technology Dependency

The risk decreased to rare probability and minor impact. All dependencies have been integrated and validated over multiple sprints.

Not Enough Buffer Planned

The risk decreased to unlikely probability and minor impact. The project is on schedule.

LLM Hallucination

The risk remains at moderate probability and extreme impact. Ongoing test runs continue to produce stochastic outputs. The evaluation pipeline monitors for hallucinations through field-level mismatch detection, but the risk of undetected edge cases persists.

LLM API Cost Overrun

The risk escalated to likely probability and moderate impact. Evaluation runs involving multiple models and repeated stochastic tests over the full test suite generate substantial token volume. Active cost monitoring is applied throughout this sprint.

8.3.8 Risk Matrix in Sprint 7

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.10: Sprint 7; Risk Matrix

Misconfiguration of Architecture

The risk remains at rare probability and moderate impact. The system is functionally complete and configuration is stable. This residual classification reflects the inherent risk present in any deployed multi-component system.

Underestimation of Man-Hours

The risk decreased to rare probability and minor impact. Remaining work is primarily documentation finalisation with a well-defined scope.

Bugs and Failures

The risk decreased to moderate probability and major impact. Active development has concluded. Remaining bugs are likely to be minor edge cases surfaced during final evaluation runs rather than systemic failures.

Agent Coordination Failure

The risk decreased to unlikely probability and moderate impact. Inter-agent coordination is stable. The primary remaining risk is edge-case failures under atypical or adversarial prompt inputs.

Scope Creep

The risk decreased to rare probability and minor impact.

Evaluation Bias and Ground Truth Quality

The risk escalated to likely probability and extreme impact. Sprint 7 is the primary sprint for thesis evaluation analysis. The risk that the ground truth dataset contains systematic gaps or that the evaluation methodology introduces bias is at its highest, as the conclusions drawn from the metrics directly shape the academic contribution of the thesis.

Unclear or Changing Requirements

The risk remains at rare probability and minor impact.

Technology Dependency

The risk remains at rare probability and minor impact.

Not Enough Buffer Planned

The risk remains at unlikely probability and minor impact.

LLM Hallucination

The risk decreased to moderate probability and major impact. Final evaluation runs continue to produce stochastic outputs, but the evaluation engine now provides comprehensive field-level coverage for detecting hallucinations as measurable mismatch events.

LLM API Cost Overrun

The risk remains at moderate probability and moderate impact. Final evaluation runs require additional API calls, but cumulative cost is monitored and expected to remain within acceptable bounds.

8.3.9 Risk Matrix in Sprint 8

Risk Categorisation / Legend:

- Blue: Project Planning Risks
- Green: Technical Risks



Figure 8.11: Sprint 8; Risk Matrix

Misconfiguration of Architecture

The risk remains at rare probability and moderate impact. The architecture is finalised; this residual classification reflects the operational nature of the deployed pipeline.

Underestimation of Man-Hours

The risk remains at rare probability and minor impact. All major work items are complete.

Bugs and Failures

The risk decreased to unlikely probability and major impact. No new development is occurring. The remaining risk is limited to previously undetected defects that could surface during final submission testing.

Agent Coordination Failure

The risk decreased to rare probability and moderate impact. The system is stable and no new coordination logic is being introduced. This residual classification reflects the live operational state of the pipeline.

Scope Creep

The risk remains at rare probability and minor impact.

Evaluation Bias and Ground Truth Quality

The risk decreased to moderate probability and major impact. The thesis evaluation analysis is substantially complete. The remaining concern concerns the adequacy of the current test suite as a representative sample of real-world onboarding complexity.

Unclear or Changing Requirements

The risk remains at rare probability and minor impact.

Technology Dependency

The risk remains at rare probability and minor impact.

Not Enough Buffer Planned

The risk remains at unlikely probability and minor impact.

LLM Hallucination

The risk decreased to unlikely probability and major impact. Hallucination detection via the evaluation pipeline is now comprehensive. Remaining exposure is limited to edge cases outside the current test suite's coverage.

LLM API Cost Overrun

The risk decreased to unlikely probability and moderate impact. API usage is winding down as development and evaluation are complete. Final documentation runs require only minimal token consumption.

Chapter 9

Personal Reports

9.1 Arnel Veladzic

The following sections reflect on my personal experience contributing to the Bachelor Thesis, with a focus on project coordination, architectural development and the transition from the Study Thesis foundation to a fully automated multi-agent system.

9.1.1 What Went Well

The iterative approach proved essential from the outset. What was not anticipated was the degree of complexity introduced by the new Atfinity sandbox tenant based on the Bank of Luxembourg’s configuration. Compared to the simplified demo tenant used in the Study Thesis, this deployment exposed a significantly larger schema: more ontologies, more fields, stricter enumeration constraints and conditional validation logic that was not transparent from the API documentation alone. Navigating this unfamiliar territory incrementally was the only realistic path forward.

The project also required a significant mid-course pivot. The original plan was to design and compare three architectural variants, directly mirroring the Study Thesis methodology. Early-Mid-project, it became clear that the iterative and hybrid approaches were structurally incompatible with multi-party onboarding: the instance-identifier tracking problem described in Chapter 5 made these designs infeasible regardless of implementation quality. Rather than producing three shallow variants, the team redirected its effort toward developing the Multi-agent Preemptive Single-Shot architecture with the depth needed to solve the problem correctly. This was a difficult decision but proved to be the right one.

9.1.2 Areas for Improvement

In future projects, I would prioritise establishing a basic evaluation pipeline as soon as the first end-to-end prototype is functional, rather than deferring it until the system is feature-complete. In this project, systematic testing began relatively late in the Construction phase. Having even a minimal test harness in place earlier would have allowed architectural problems to be detected and corrected sooner, reducing the cost of late-stage redesign. Building the measurement infrastructure in parallel with the system is the single change that would have had the greatest impact on iteration speed.

9.1.3 Personal Highlights

The primary highlight was designing and deploying a production-oriented multi-agent AI system that solves a real problem for a real company. Moving from Study Thesis prototype variants to a fully automated, CI/CD-integrated pipeline demonstrated that the architecture had matured into a genuinely deployable system. This project strengthened my capabilities in system architecture, technical leadership and the disciplined application of emerging AI technologies within a commercially relevant context.

9.2 Fabio Gomes Silva

The following sections reflect on my personal experience contributing to the Bachelor Thesis, with a focus on evaluation methodology, quality measurement and the development of the deterministic testing framework.

9.2.1 What Went Well

The automated testing framework developed for this thesis became the primary driver of architectural decision-making throughout the project. By designing a rigorous, deterministic evaluation pipeline with precision, recall and F1 metrics computed at the field level, the team was able to assess architectural variants objectively rather than relying on subjective impressions. The test suite, comprising 12 test cases across three ground truth onboarding scenarios and four prompt archetypes, provided sufficient coverage to expose meaningful differences in model behaviour, field extraction quality and robustness to prompt variability.

The collaboration with Arnel mostly remained productive throughout all eight sprints. While he focused on architecture and implementation, my contribution centred on evaluation methodology, test case quality and the interpretation of results. The deterministic evaluation engine, which resolves complex entity aliasing, handles auto-calculated field exclusion and applies collision-proof field scoring, represents a technically substantial contribution that transformed evaluation from a qualitative exercise into a reproducible, quantitative process.

9.2.2 Areas for Improvement

Evaluation methodology benefits significantly from external review and earlier feedback from the supervisor on the test case design would have allowed the team to identify coverage gaps sooner. In future projects, I will seek guidance on measurement strategies earlier, rather than finalising the framework independently before presenting it for review.

9.2.3 Personal Highlights

The most satisfying outcome of this project was the discovery that model selection has a fundamental and measurable effect on multi-agent system behaviour. Observing that specific models consistently outperformed others on structured field extraction tasks, particularly in their ability to correctly invoke MCP tools under complex ontological constraints, provided concrete evidence that model selection is an architectural decision, not merely an operational one.

Working with non-deterministic systems and designing a reproducible evaluation methodology for them required careful thought about statistical robustness, entity resolution and the semantics of field matching within a proprietary CMS data model. The stochastic repetition mechanism and the collision-proof field evaluation protocol were direct responses to challenges encountered during testing and represent non-obvious engineering contributions. Knowing that the evaluation results provided the empirical foundation for the thesis conclusions gave my work direct academic and practical significance beyond the testing role itself.

Bibliography

- [1] Anthropic. Claude. Used as a support tool for drafting, language refinement and documentation assistance. No content was generated without human review. URL: <https://claude.com/product/overview>.
- [2] Atfinity. Atfinity api documentation. Technical API reference, [last access: 2025-11-03]. URL: <https://api-docs.atfinity.io/#intro>.
- [3] Atfinity. Atfinity api documentation — case creation with predefined objects. Technical API reference, [last access: 2026-05-20]. URL: <https://api-docs.atfinity.io/#0f31ac27-2a64-4c92-a650-8ec2ad0623cd>.
- [4] Atfinity AG. Atfinity – intelligent process automation for financial institutions. Official company website, [last access: 2025-12-16]. URL: <https://www.atfinity.swiss>.
- [5] Simon Brown. The c4 model for visualizing software architecture. Software architecture visualization method, [last access: 2025-12-03]. URL: <https://c4model.com/>.
- [6] European Parliament and Council of the European Union. Regulation (eu) 2016/679 (general data protection regulation). Official Journal of the European Union, [last access: 2026-05-20]. URL: <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679>.
- [7] European Parliament and Council of the European Union. Regulation (eu) 2024/1689 of the european parliament and of the council laying down harmonised rules on artificial intelligence (artificial intelligence act). Official Journal of the European Union, L series, 2024, [last access: 2026-05-20]. URL: <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32024R1689>.
- [8] Financial Action Task Force (FATF). The fatf recommendations: International standards on combating money laundering and the financing of terrorism & proliferation. Updated standards, [last access: 2025-11-15]. URL: <https://www.fatf-gafi.org/en/topics/fatf-recommendations.html>.
- [9] IBM. What are hierarchical ai agents? IBM Think Topics, [last access: 2026-05-18]. URL: <https://www.ibm.com/think/topics/hierarchical-ai-agents>.
- [10] IBM. What is a multi-agent system? IBM Think Topics, [last access: 2026-05-18]. URL: <https://www.ibm.com/think/topics/multiagent-system>.
- [11] LibreChat. Librechat documentation. Official software documentation, [last access: 2025-11-10]. URL: <https://www.librechat.ai/docs>.
- [12] Model Context Protocol. Model context protocol: Getting started. Technical specification and documentation, [last access: 2025-11-27]. URL: <https://modelcontextprotocol.io/docs/getting-started/intro>.
- [13] n8n. Ai agent architecture patterns. Published 7 May 2026, [last access: 2026-06-09]. URL: <https://blog.n8n.io/ai-agent-architecture-patterns/>.
- [14] OpenAI. Chatgpt. Used as a support tool for drafting, language refinement and documentation assistance. No content was generated without human review. URL: <https://chatgpt.com/>.
- [15] OpenAI. Function calling and other api updates. Official announcement of function calling for GPT models, [last access: 2026-05-20]. URL: <https://openai.com/index/function-calling-and-other-api-updates/>.

- [16] OpenAI. GPT-4 Technical Report, 2023. [arXiv:2303.08774](#).
- [17] Swiss Financial Market Supervisory Authority (FINMA). Finma circular 2023/1: Operational risks and resilience — banks. Replaces FINMA Circular 2008/21 on operational risks; includes provisions on outsourcing and cloud services, [last access: 2026-05-20]. URL: <https://www.finma.ch/en/regulation/circulars/>.
- [18] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2nd edition, 2009.
- [19] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation, 2023. [arXiv:2308.08155](#).
- [20] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models, 2023. [arXiv:2210.03629](#).

List of Figures

1	How it Works.	v
2	Overall average precision, recall and F1-score per AI model, aggregated across all tested ground-truth scenarios.	vi
2.1	Use Case Diagram: Multi-agent Preemptive Single-Shot	12
3.1	MCP as a universal interface layer between LLM clients and backend systems [12]	18
5.1	System Context Diagram	24
5.2	C4 Container Diagram: Multi-agent Preemptive Single-Shot Architecture	29
5.3	Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Background phase: Sync Container startup and schema synchronisation	37
5.4	Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Preview phase . . .	39
5.5	Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Create phase, Steps 0 to 2: cache verification	42
5.6	Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Create phase, Steps 3 and 4: parallel extraction and payload assembly	44
5.7	Sequence diagram of the Multi-agent Preemptive Single-Shot pipeline, Create phase, Step 5: atomic submission with self-healing	46
5.8	Example <code>predefined_objects</code> payload for POST <code>/api/1/cases</code> [3]	51
6.1	Directory structure of the <code>onto-db</code> shared Docker volume after initial synchronisation . .	68
7.1	Preview output returned by <code>preview_onboarding</code> : organised client data, matched process, business partner type and entity-role plan presented to the Relationship Manager before case creation.	86
7.2	Case creation confirmation returned by <code>create_client_onboarding</code> : Atfinity case identifier, parties created with their assigned roles and total field count	87
7.3	The corresponding case in Atfinity's CMS after initial creation and subsequent field updates via <code>update_case_field</code> . Field values differ from the creation output because post-creation corrections were applied.	87
7.4	Average pipeline duration (X-axis) versus average F1-score (Y-axis) per model and ground-truth scenario. Bubble area is proportional to the average cost per test case. Haiku TEST2 reflects the email archetype only.	92
7.5	Atfinity's HTTP 400 rejection on the fifth and final retry attempt for a GPT-5.4-mini evaluation run. The model consistently formatted currency-amount fields as nested JSON objects containing <code>decimal_number</code> , <code>unit</code> and <code>item</code> keys rather than the plain decimal scalar that Atfinity's API requires. This type mismatch triggered a rejection on every submission attempt.	103
7.6	Self-healing loop execution for GPT-5.4-mini alongside the contrasting GPT-5.4 result for the identical test case. When the Orchestrator routed the rejected fields back to the Person Agent with the full error context, GPT-5.4-mini returned <code>None</code> for the first field (stripping it rather than correcting it) and an integer 0 for the second, neither satisfying Atfinity's type constraint. GPT-5.4, given the same prompt and schema, required no correction at all and succeeded on its first submission attempt.	104
7.7	Average F1-score per ground-truth scenario for each evaluated model, with each bar representing the mean across the four conversational archetypes (Structured, Messy, Dictation and Email). The dashed line marks the NFR-4 acceptance threshold of 85%.	105

7.8	Precision-recall scatter plot per model and test case across the full evaluation suite. Each point represents one of the twelve test cases per model. The three Haiku TEST2 pipeline failures are excluded as their values fall outside the plotted range.	107
7.9	Average precision and recall per model and ground-truth scenario, where each bar represents the mean across all conversational archetypes within that scenario. Haiku TEST2* includes the email archetype only; the three pipeline failures are excluded. Complements Figure 7.8: where the scatter plot shows one data point per archetype (e.g. TEST1_STRUCT, TEST1_DICT), this chart collapses those points into a single scenario-level average, making the per-scenario trend directly readable.	108
7.10	Mean F1-score and run-to-run standard deviation per test case across all three evaluated models.	109
7.11	Total correction cycles per test case and model, summed over five independent runs. . . .	110
7.12	End-to-end pipeline duration per test case and model, grouped by ground-truth scenario. . . .	111
7.13	Average cost per case versus average F1-score per model and ground-truth scenario. . . .	112
7.14	Average pipeline duration versus average F1-score per model and ground-truth scenario. . .	113
7.15	Average input token consumption per model and ground-truth scenario.	115
7.16	Average output token consumption per model and ground-truth scenario.	116
7.17	Average input tokens versus average output tokens per model and ground-truth scenario, with bubble area proportional to cost.	117
8.1	Final Project Plan	124
8.2	Time spent per sprint of the whole team	127
8.3	Time spent per team member for whole project	128
8.4	Sprint 1; Risk Matrix	131
8.5	Sprint 2; Risk Matrix	133
8.6	Sprint 3; Risk Matrix	135
8.7	Sprint 4; Risk Matrix	137
8.8	Sprint 5; Risk Matrix	139
8.9	Sprint 6; Risk Matrix	141
8.10	Sprint 7; Risk Matrix	143
8.11	Sprint 8; Risk Matrix	145
A.1	Onboarding Step 1 - The RM submits a complex, unstructured meeting note as a natural-language prompt. The input includes compliance deficiencies, AML flags, identified entities and associated party details for the Margrethus AG trust structure.	157
A.2	Onboarding Step 2 - The system invokes <code>preview_onboarding</code> and returns an initial structured preview. The identified workflow is “Compliance Pre-Check” (Process ID: 32) with a Prospect outcome - an incorrect match that the RM will subsequently reject.	157
A.3	Onboarding Step 3 - The structured entity extraction from the preview. The system has correctly identified the corporate entity Margrethus AG and three associated persons (Julia Maria JUBELPETER as Director, Maria MORGARTEN as a deceased Beneficiary with an unresolved succession warning and Peter MARBURGER as Signatory).	158
A.4	Onboarding Step 4 - The RM rejects the preview (“no”) and indicates that the workflow selection is incorrect. The system acknowledges and requests clarification on the intended workflow. The RM responds: “i don’t know but this workflow is wrong.”	159
A.5	Onboarding Step 5 - The system autonomously invokes <code>list_onboarding_processes</code> to identify the best-matching workflow. It selects “Account Onboarding” (Process ID: 41) with 95% confidence. After RM confirmation, a corrected preview is generated showing the Account Onboarding workflow with Business Partner Type: Trust.	160
A.6	Onboarding Step 6 - The RM provides an additional instruction before approval: “margrethus ag should be also a person in this case.” The system re-runs <code>preview_onboarding</code> incorporating this structural change.	161
A.7	Onboarding Step 7 - The updated preview now reflects the structural change: Margrethus AG is listed both as a corporate entity and as an additional Person instance within the case, as indicated by the “Also added as a Person instance in the case” annotation.	161
A.8	Onboarding Step 8 - The RM approves the updated preview (“yes”). The system invokes <code>create_client_onboarding</code> and confirms successful creation: Case #1110 under the Account Onboarding process (ID: 41), with 312 fields populated across 13 instances.	162

A.9	Onboarding Step 9 - Post-creation case management. The system proactively identifies a role inconsistency and offers corrections. The RM requests option 1: correct Julia Maria JUBELPETER's role from Trustee to Director. The system calls set_instance_roles and confirms the role update on instance 2779.	163
A.10	Onboarding Step 10 - The RM requests that Margrethus AG also receive the Trust role. The system calls add_instance_role and confirms: Margrethus AG (instance 2774) now holds the Trust role and the auto-created placeholder person (instance 2781) has been automatically removed by the CMS.	163

List of Tables

6.1	Key Python libraries used in the Multi-agent Preemptive Single-Shot architecture	67
7.1	User Story Implementation Status	84
7.2	Functional Requirements Implementation Status	84
7.3	Non-Functional Requirements Verification Status	84
7.4	LLM configuration summary for the comparative evaluation	93
7.5	GPT-5.4: full evaluation results across all twelve test cases. All metric values are averages over five independent runs per test case. F1 = average F1-score (%); Prec. = average precision (%); Rec. = average recall (%); σ = standard deviation across the five runs; Cost = average USD cost per run; Dur = average pipeline duration in seconds; Ret = total correction cycles per test case across five runs.	94
7.6	GPT-5.4: input and output token consumption and pipeline duration per test case, each averaged over five repeated runs. In:Out = integer ratio of input to output tokens. Duration reflects the full end-to-end pipeline from prompt submission to confirmed case creation.	96
7.7	Claude Sonnet 4.6: full evaluation results across all twelve test cases. F1 = F1-score (%); Prec = Precision (%); Rec = Recall (%); σ = standard deviation across repeated runs; Cost = average USD; Dur = average duration in seconds; Ret = total correction cycles per test case across five runs.	97
7.8	Claude Sonnet 4.6: input and output token consumption and pipeline duration per test case, averaged across three to five completed runs depending on the test case. Duration reflects the full end-to-end pipeline latency.	99
7.9	Claude Haiku 4.5: full evaluation results. TEST2_STRUCT, TEST2_MESSY and TEST2_DICT produced complete LLM errors; no case was created and all metrics are zero. F1 = F1-score (%); Prec = Precision (%); Rec = Recall (%); σ = standard deviation; Cost = average USD; Dur = average duration in seconds; Ret = total correction cycles per test case across five runs (25 = correction budget exhausted on every run).	100
7.10	Claude Haiku 4.5: input and output token consumption and pipeline duration per test case, averaged over five repeated runs. The three TEST2 error rows log no token metrics as the pipeline failed before Ontology Agent completion.	102
7.11	Average precision and recall per model across all twelve test cases.	106
7.12	NFR compliance assessment per evaluated model. Aggregate averages are reported for context; individual test case values are available in Tables 7.5–7.9.	106
7.13	Total correction cycles per test case and model, summed over five independent runs.	111
7.14	Cross-model operational metrics per scenario.	114
7.15	Average input and output token consumption per scenario and model, per pipeline execution.	114
8.1	Project roles and team members	121
8.2	Risks and their mitigations (colour matching to risk matrix)	130

Appendices

Appendix A

Examples of Working Client Onboardings

This appendix provides supplementary material referenced throughout the thesis. It comprises a complete multi-turn onboarding walkthrough, a demonstration of the autonomous self-healing mechanism, the full Mermaid source code for the system sequence diagrams and the complete Ontology Agent system prompt template used for field extraction (Section ??).

A.1 Example of a Full Client Onboarding

The following screenshots document a complete end-to-end onboarding interaction using ground truth case 1204 (Margrethus AG), a complex Swiss trust structure involving a corporate entity, five associated persons and cross-jurisdictional compliance considerations. The scenario demonstrates the full iterative dialogue: initial preview, rejection with correction, process re-identification, entity adjustment, case creation and post-creation field management.

The RM (user “arnel”) submits unstructured meeting notes containing compliance deficiencies, flagged AML sensitivities and partial personal data. The system processes the input, extracts structured previews, responds to corrections and ultimately creates the case in Atfinity’s CMS with all required instances and fields populated.

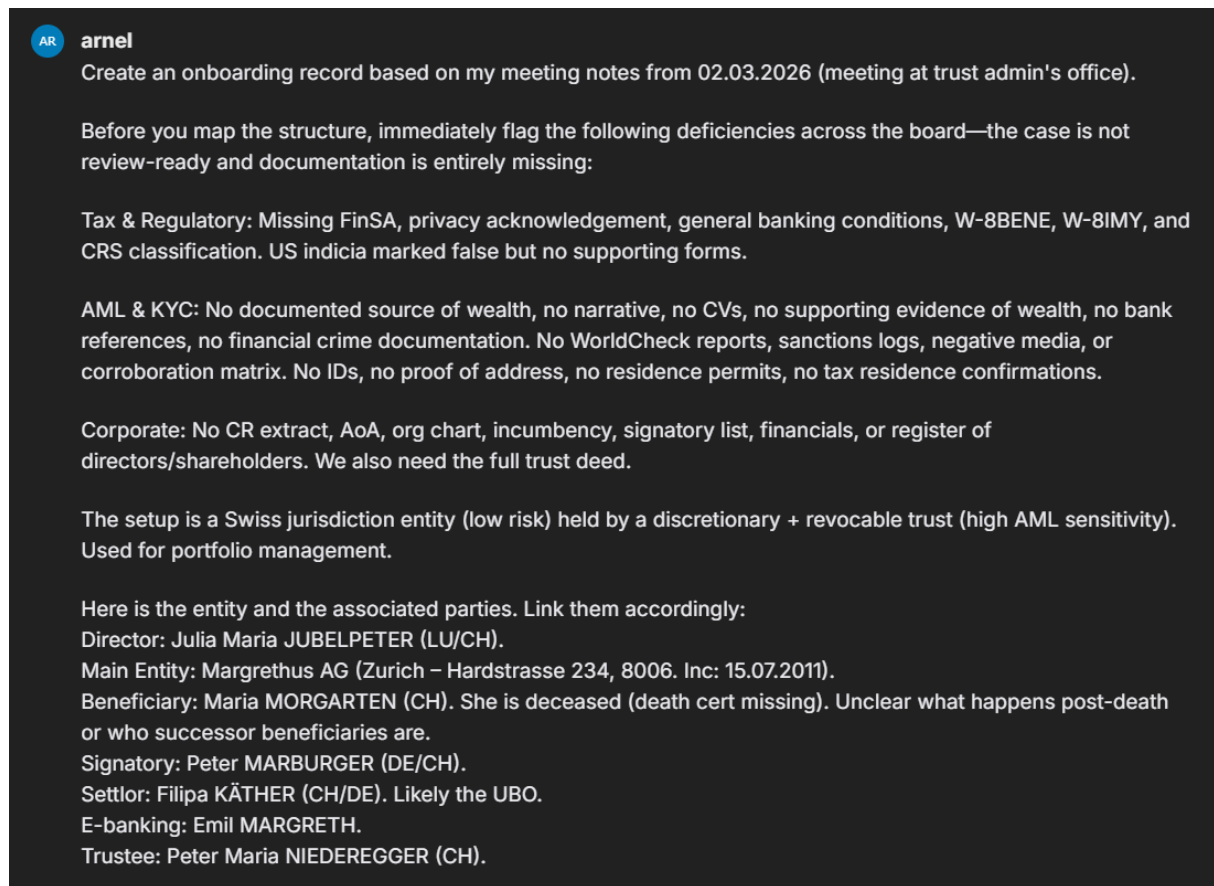


Figure A.1: Onboarding Step 1 - The RM submits a complex, unstructured meeting note as a natural-language prompt. The input includes compliance deficiencies, AML flags, identified entities and associated party details for the Margrethus AG trust structure.

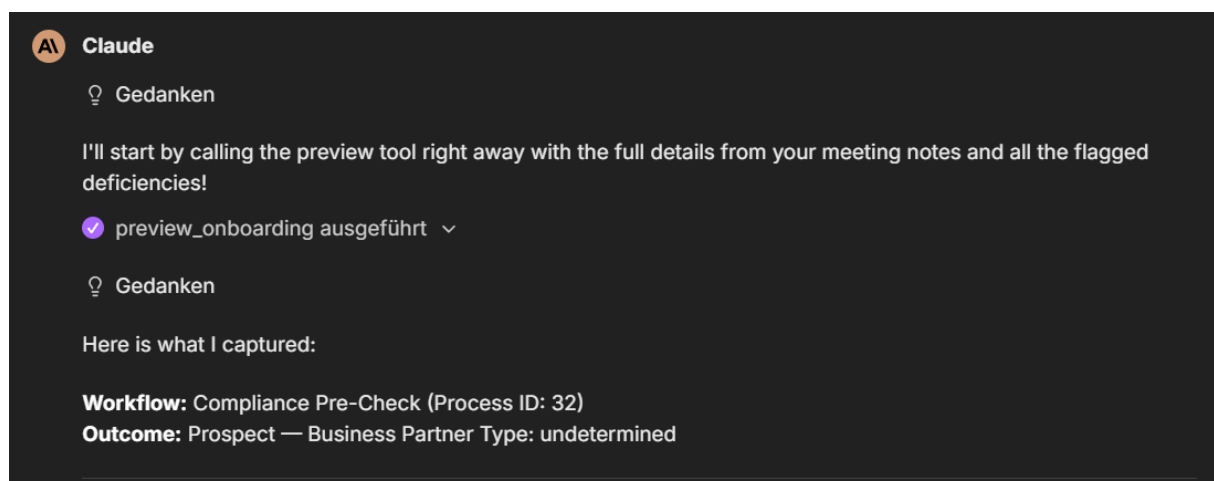


Figure A.2: Onboarding Step 2 - The system invokes `preview_onboarding` and returns an initial structured preview. The identified workflow is “Compliance Pre-Check” (Process ID: 32) with a Prospect outcome - an incorrect match that the RM will subsequently reject.

 **Entity — Margrethus AG**

- Address: Hardstrasse 234, 8006 Zurich, Switzerland
- Incorporated: 15.07.2011
- Structure: Swiss jurisdiction entity (*low risk*), held by a **discretionary & revocable trust** (*high AML sensitivity*)
- Purpose: Portfolio management

 **Person 1 — Julia Maria JUBELPETER**

- Role: **Director**
- Nationalities: LU / CH

 **Person 2 — Maria MORGARTEN** 

- Role: **Beneficiary**
- Nationality: CH
- **Deceased** — death certificate missing
-  Unclear what happens post-death / successor beneficiaries unidentified

 **Person 3 — Peter MARBURGER**

- Role: **Signatory**
- Nationalities: DE / CH

Figure A.3: Onboarding Step 3 - The structured entity extraction from the preview. The system has correctly identified the corporate entity Margrethus AG and three associated persons (Julia Maria JUBELPETER as Director, Maria MORGARTEN as a deceased Beneficiary with an unresolved succession warning and Peter MARBURGER as Signatory).

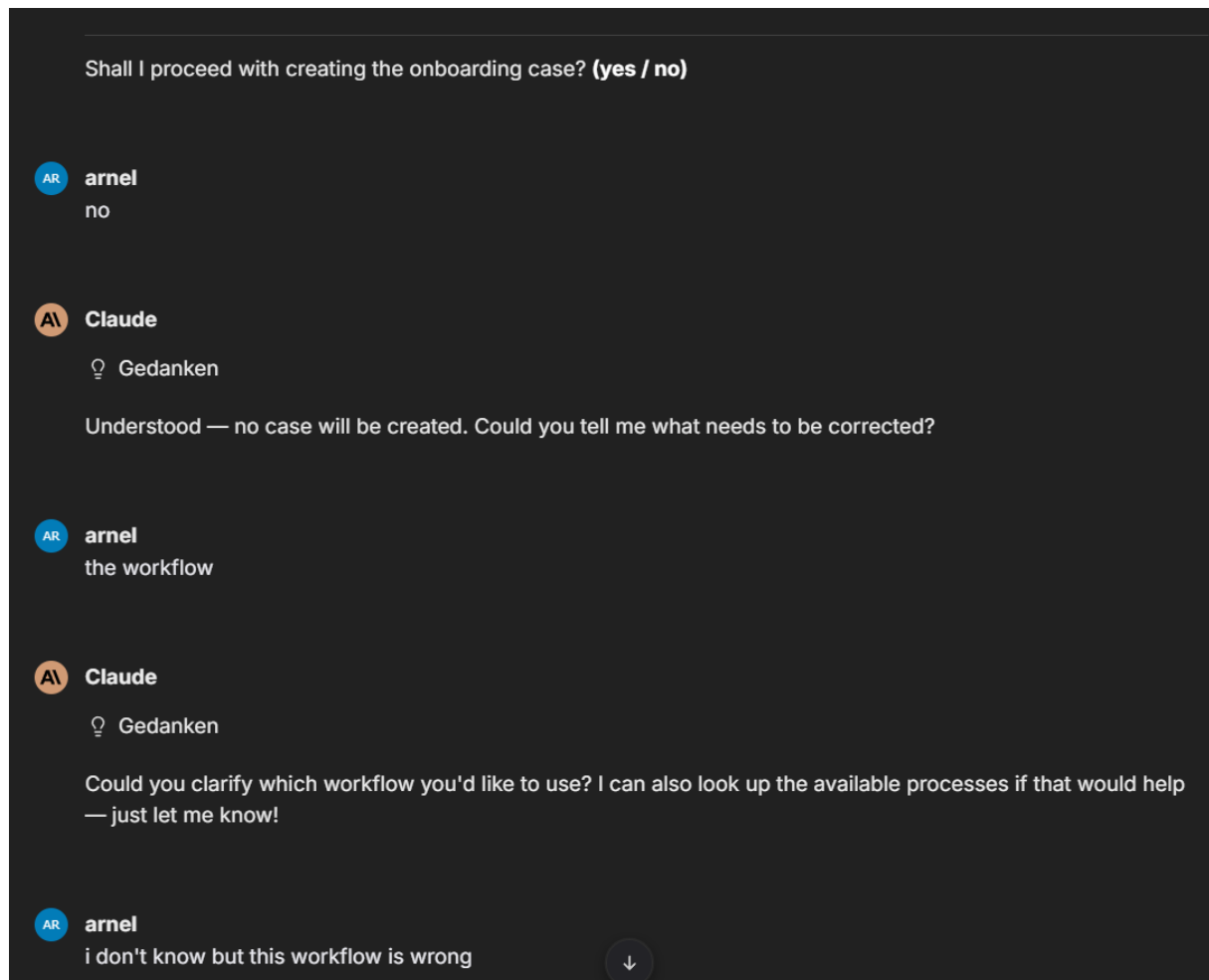


Figure A.4: Onboarding Step 4 - The RM rejects the preview (“no”) and indicates that the workflow selection is incorrect. The system acknowledges and requests clarification on the intended workflow. The RM responds: “i don’t know but this workflow is wrong.”

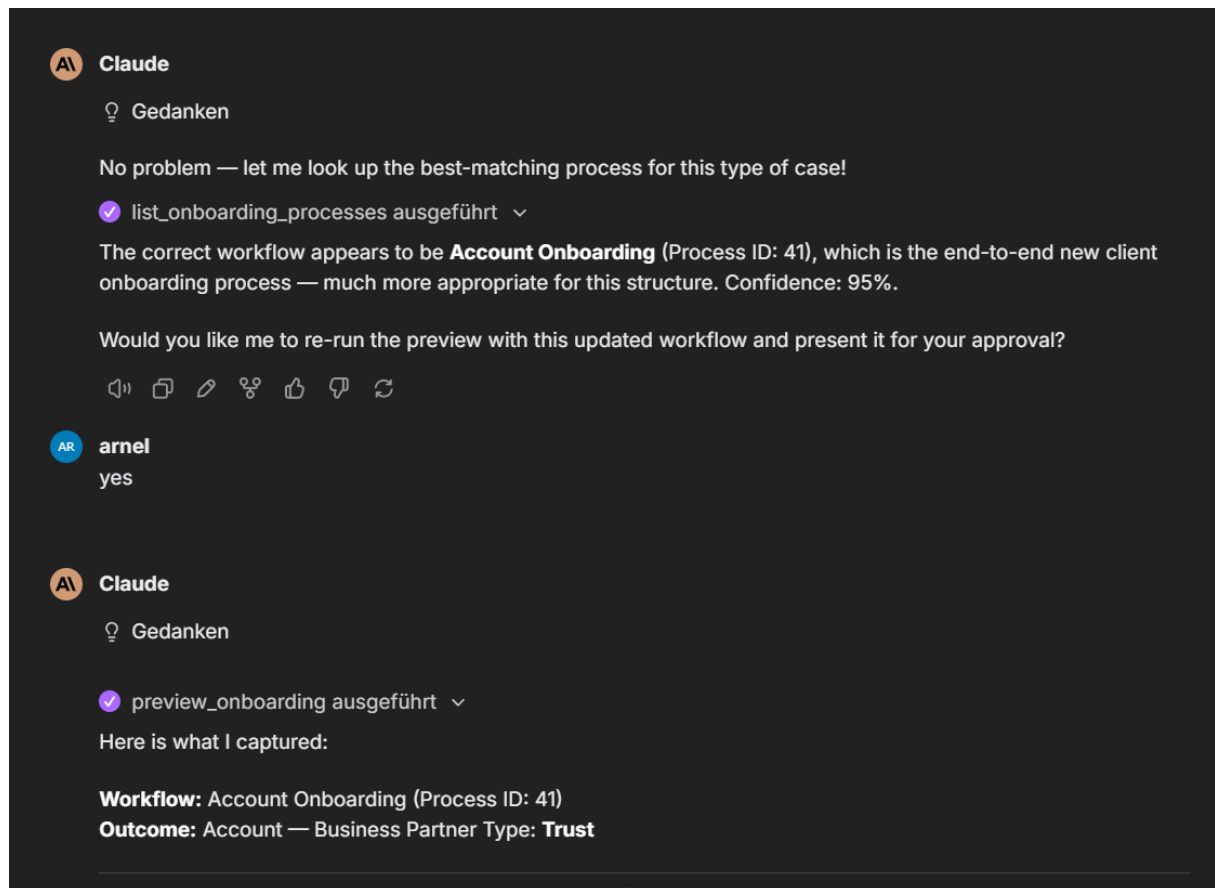


Figure A.5: Onboarding Step 5 - The system autonomously invokes `list_onboarding_processes` to identify the best-matching workflow. It selects “Account Onboarding” (Process ID: 41) with 95% confidence. After RM confirmation, a corrected preview is generated showing the Account Onboarding workflow with Business Partner Type: Trust.

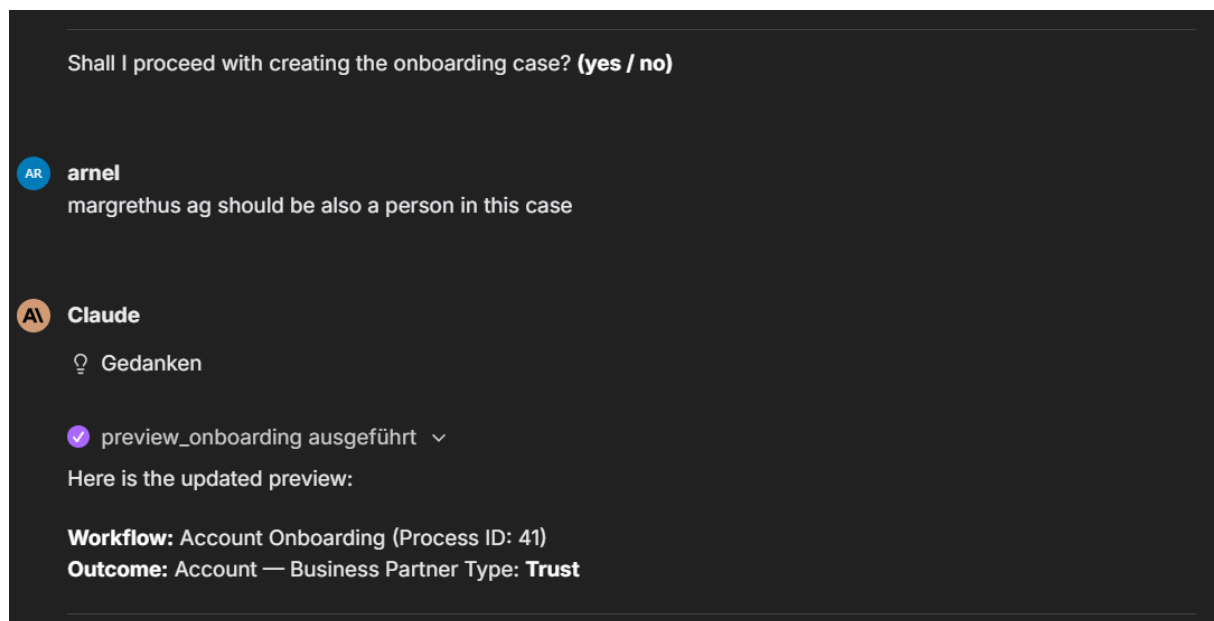


Figure A.6: Onboarding Step 6 - The RM provides an additional instruction before approval: “margrethus ag should be also a person in this case.” The system re-runs `preview_onboarding` incorporating this structural change.

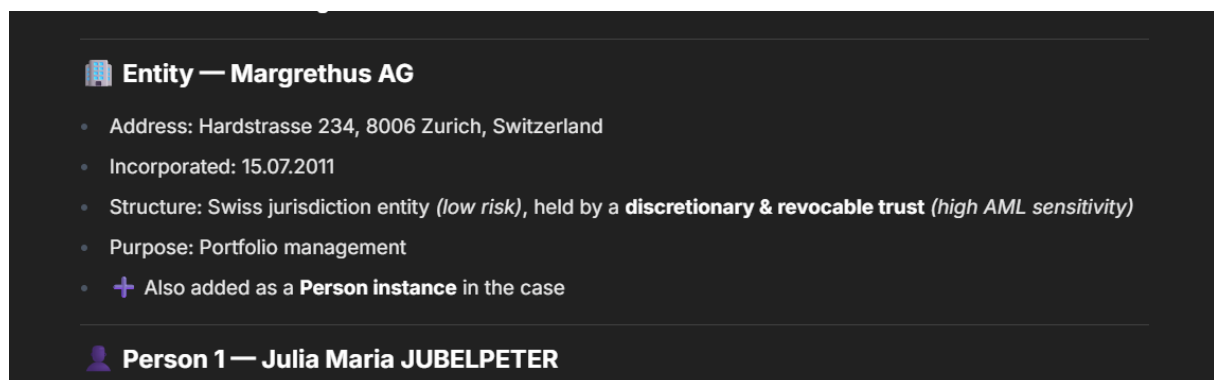


Figure A.7: Onboarding Step 7 - The updated preview now reflects the structural change: Margrethus AG is listed both as a corporate entity and as an additional Person instance within the case, as indicated by the “Also added as a Person instance in the case” annotation.

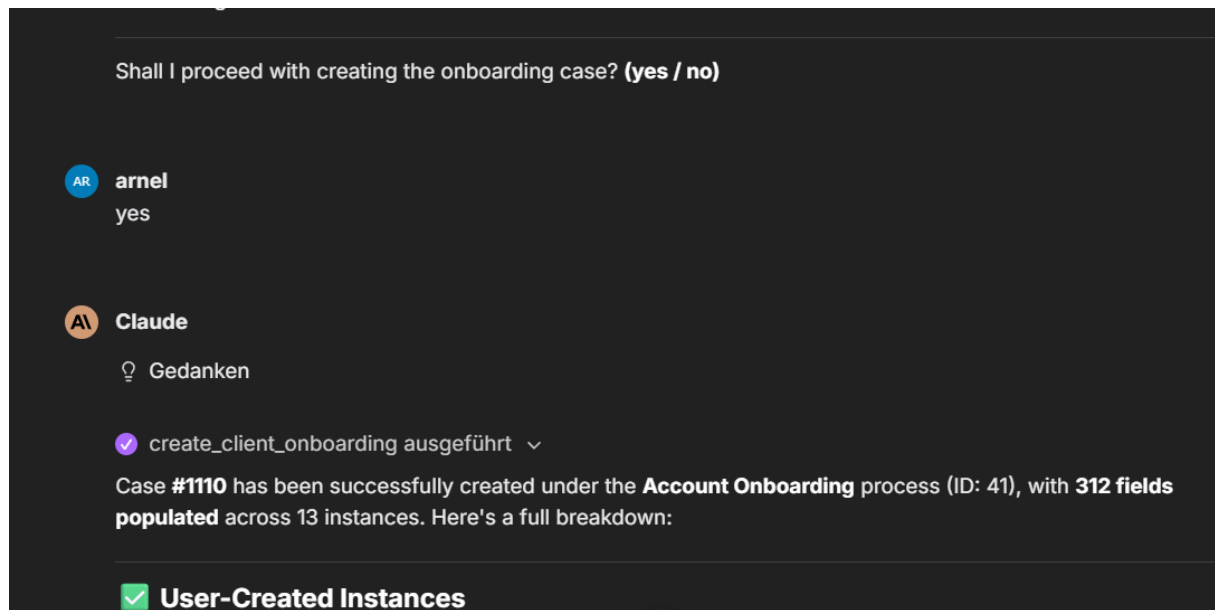


Figure A.8: Onboarding Step 8 - The RM approves the updated preview (“yes”). The system invokes `create_client_onboarding` and confirms successful creation: Case #1110 under the Account Onboarding process (ID: 41), with 312 fields populated across 13 instances.

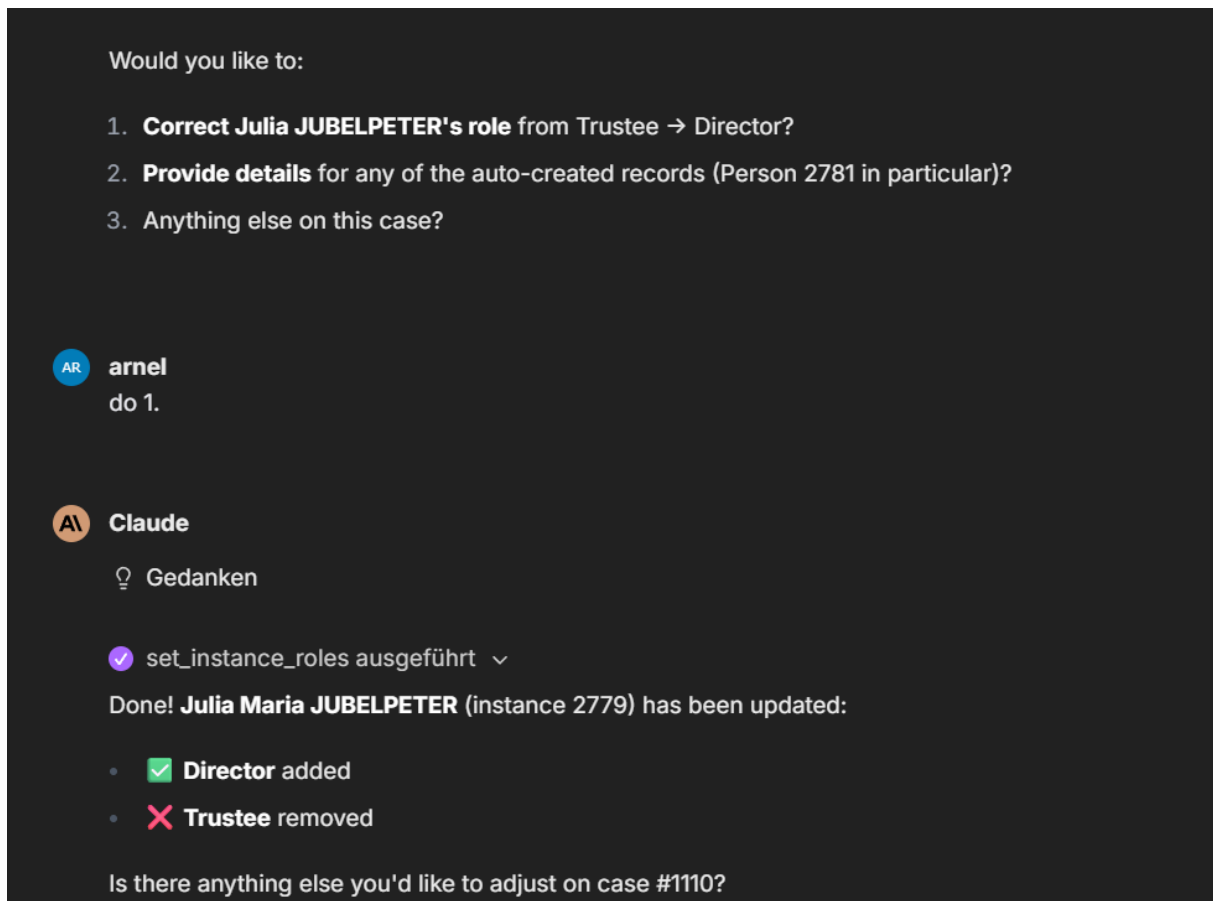


Figure A.9: Onboarding Step 9 - Post-creation case management. The system proactively identifies a role inconsistency and offers corrections. The RM requests option 1: correct Julia Maria JUBELPETER's role from Trustee to Director. The system calls `set_instance_roles` and confirms the role update on instance 2779.

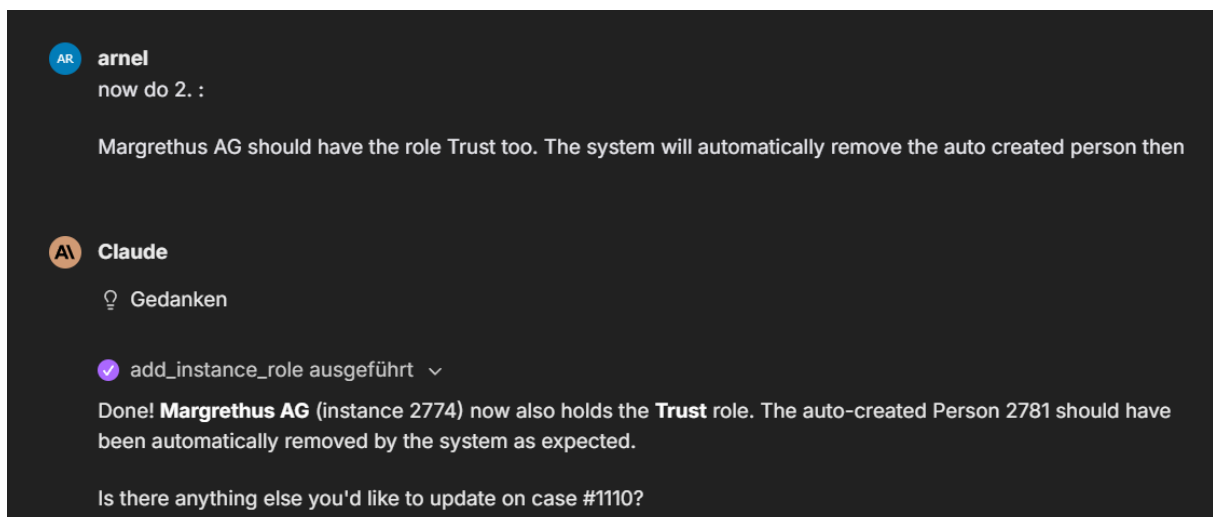


Figure A.10: Onboarding Step 10 - The RM requests that Margrethus AG also receive the Trust role. The system calls `add_instance_role` and confirms: Margrethus AG (instance 2774) now holds the Trust role and the auto-created placeholder person (instance 2781) has been automatically removed by the CMS.

A.2 Mermaid Source Code - Full Sequence Diagram

The following Mermaid source code defines the complete system sequence diagram covering all three interaction phases: the background startup and schema synchronisation, the preview flow and the create flow. To render the diagram interactively, copy the code block below and paste it into <https://mermaid.live/>.

```
%%{init: {'theme': 'default', 'themeVariables': {'fontSize': '20px',
'fontFamily': 'arial'}}}%
sequenceDiagram
    autonumber
    actor RM
    participant LC as LibreChat

    box rgb(240, 248, 255) Orchestration Hub
        participant Orch as Orchestrator
        participant CACHE as Preview Cache
        participant Router as Onto. Router
        participant SchemaDB as Schema-DB
    end

    box rgb(255, 245, 235) Process Determination
        participant PDA as Process Agent
        participant MemPDA as PDA Context
    end

    box rgb(245, 255, 240) Prompt Organization
        participant POA as Prompt Organizer
    end

    box rgb(245, 240, 255) Ontology Agents [1..N]
        participant OA_A as OA: Account
        participant OA_P as OA: Person
        participant OntoDB as Onto-DB
    end

    participant API
    participant Atf as Atfinity
    participant SYNC as Sync

    %% BACKGROUND
    rect rgb(240, 210, 255)
    Note over LC, SYNC: BACKGROUND -- Startup and schema synchronisation (runs once, exits)

    Note over SYNC: Reads manifest, builds image,
    Note over SYNC: deploys one container per ontology
    SYNC->>OA_A: Deploy OA Account
    SYNC->>OA_P: Deploy OA Person

    SYNC->>Atf: GET processes
    Atf-->>SYNC: Process configs
    SYNC->>MemPDA: Write processes_mapping.yaml

    SYNC->>Atf: GET ontologies
    Atf-->>SYNC: Ontology schemas
    SYNC->>OntoDB: Write ontology_*.yaml

    SYNC->>Router: Write orch_context.yaml
    end

    %% PREVIEW FLOW
    Note over RM, Atf: PREVIEW -- RM reviews structured data before any case is created

    RM->>LC: Natural language prompt
    LC->>Orch: preview_onboarding(prompt)
    Note over Orch: All 4 steps run on first call.
    Note over Orch: On correction: Step 1 and 4 always re-run,
    Note over Orch: Step 2 and 3 skipped unless rerun_process=True

    Note over Orch, POA: Step 1 -- Prompt Reorganization (always runs)
    Orch->>POA: POST /organize
    POA->>POA: LLM restructures prompt into labelled entity sections
    POA-->>Orch: organised_prompt
```

```

Note over Orch, MemPDA: Step 2 -- Process Determination (skipped on corrections)
Orch->>PDA: POST /determine_process
PDA->>MemPDA: Load process list
MemPDA-->>PDA: Processes with intents
PDA->>PDA: LLM selects best-matching process
PDA-->>Orch: ProcessMatch {process_id, outcome_ontology, confidence}

Note over Orch, OntoDB: Step 3 -- Outcome Extraction (skipped with Step 2)
Orch->>OA_A: POST /process (outcome agent)
OA_A->>OntoDB: Read Account schema
OntoDB-->>OA_A: Account schema
OA_A->>OA_A: LLM extracts fields
OA_A-->>Orch: Account fields + business_partner_type

Note over Orch, Router: Step 4 -- Ontology Routing (always re-runs)
Orch->>Router: identify_ontologies(organised_prompt)
Router->>Router: LLM assigns entities, roles, instance prompts
Router-->>Orch: [{key, roles, instance_prompt}]

Orch->>CACHE: Store all preview results
Orch-->>LC: Preview {process, outcome, ontologies}
LC-->>RM: Displays structured preview

%% CREATE FLOW
Note over RM, Atf: CREATE -- Cache reused from preview, redundant LLM calls skipped

RM->>LC: Approves preview
LC->>Orch: create_client_onboarding(prompt)
Note over Orch: Checks cache for each step

Note over Orch, POA: Step 0 -- Organised Prompt
Orch->>CACHE: get_pending()
alt Cache hit
    CACHE-->>Orch: organised_prompt (reused)
else Cache miss
    Orch->>POA: POST /organize
    POA-->>Orch: organised_prompt
end

Note over Orch, MemPDA: Step 1 -- Process Match
alt Cache hit
    CACHE-->>Orch: ProcessMatch (reused)
else Cache miss
    Orch->>PDA: POST /determine_process
    PDA-->>Orch: ProcessMatch
end

Note over Orch, Router: Step 2 -- Ontology Routing
alt Cache hit
    CACHE-->>Orch: ontologies (reused)
else Cache miss
    Orch->>Router: identify_ontologies(prompt)
    Router-->>Orch: [{key, roles, instance_prompt}]
end

Note over Orch, OntoDB: Step 3 -- Parallel Field Extraction
Note over Orch: Each agent receives a focused instance_prompt
par Fan-out to all required agents
    Orch->>OA_A: POST /process (outcome)
    Orch->>OA_P: POST /process (instance prompt)
    OA_A->>OntoDB: Read Account schema
    OntoDB-->>OA_A: Account schema
    OA_A->>OA_A: LLM extracts and validates fields
    OA_P->>OntoDB: Read Person schema
    OntoDB-->>OA_P: Person schema
    OA_P->>OA_P: LLM extracts and validates fields
    OA_A-->>Orch: Account fields
    OA_P-->>Orch: Person fields
end

Note over Orch, SchemaDB: Step 4 -- Payload Assembly
Orch->>SchemaDB: get_base_schema()

```

```

SchemaDB-->>Orch: Learned payload skeleton
Orch->>Orch: Inject process_id, place outcome first,
Orch->>Orch: append domain objects with ontology key and roles

Note over Orch, Atf: Step 5 -- Atomic Submission with Self-Healing (max 5 retries)
loop Self-Healing (max 5 retries)
  Orch->>API: Submit payload
  API->>Atf: POST /api/1/cases (atomic bulk)
  Atf-->>API: 200 OK or 400 error
  API-->>Orch: SubmitResult

  alt 200 OK
    Orch->>SchemaDB: save_successful_payload()
    Orch->>CACHE: evict(prompt)
    Orch->>LC: {case_id, instances, roles}
    LC-->>RM: Onboarding complete
  else 400 -- ontology error
    Orch->>Orch: Remove failing ontology block
  else 400 -- structural error
    Orch->>Orch: Patch malformed field value
  else 400 -- semantic error
    Orch->>OA_P: POST /correct {failed_fields}
    OA_P-->>Orch: Corrected values
    Orch->>Orch: Patch corrected values into payload
  else Non-recoverable
    Orch->>LC: {status: failed, message}
    LC-->>RM: Error -- case not created
  end
end
end

```