



**Politecnico
di Torino**

Politecnico di Torino

Master's Degree in Cybersecurity

Academic Year 2025/2026

Graduation Session July 2026

Towards an autonomous CTF solver

A neuro-symbolic multi-agent approach

Supervisor

Prof. Cataldo Basile

Dott. Davide Colaiacomo

Dott.ssa Chiara Bonfanti

Candidate

Carmelo Bertolami

To my parents, for their unwavering belief in me.

Abstract

Capture the Flag (CTF) challenges are a well-established benchmark for evaluating automated cybersecurity systems. With the introduction of powerful tools such as Large Language Models (LLMs), interest in solving these challenges automatically is rapidly growing. These models present both architectural strengths and systematic limitations that hinder their practical applicability.

This work proposes and evaluates a neuro-symbolic multi-agent architecture for automatically resolving cryptographic CTF challenges. The architecture is organised into three layers: a Coordinator agent that manages the entire flow from challenge analysis to attack hypothesis; a Belief-Desire-Intention (BDI) agent that structures the reasoning, generation, validation, execution, and debugging cycle; and an embodied Explorer agent that perceives and acts within an isolated Docker sandbox. The choice of which attack to attempt is driven by two knowledge sources fused into a single ranked list of hypotheses: a CLIPS (C Language Integrated Production System) expert system that encodes cryptographic attack rules, and a Retrieval-Augmented Generation (RAG) module, reused from a previous thesis, that retrieves relevant techniques from a corpus of CTF write-ups. The primary aim is to show that the surrounding structure, rather than the model’s scale, is what makes a small local model succeed. Running such a model locally is also motivated by the goal of operational independence from external providers.

The evaluation is conducted on a dataset of cryptographic challenges spanning both attack and construction tasks, across three experimental scenarios: a direct API call without scaffolding, a commercial agent with tool-calling (OpenCode), and the system developed in this work. Qwen 2.5-Coder-7B quantised to 4-bit and GPT-4o-mini were tested across all three scenarios. Results show that both models under raw conditions reach a 0% solve rate while identifying a correct or partially correct approach in most cases (85% with Qwen and 90% with GPT-4o-mini), confirming that the lack of code execution is the primary constraint. The commercial agent, OpenCode, reaches a 10% solve rate with GPT-4o-mini and 0% with the local model. The proposed system achieves 55% with Qwen 7B and 70% when the whole pipeline runs on GPT-4o-mini instead of the local model. A failure analysis, using a taxonomy that distinguishes between implementation errors and incorrect technique selection, shows that the limiting factor is not the strategy chosen by the symbolic component but the quality of the generated code.

Table of Contents

1	Introduction	1
1.1	Context	1
1.2	The Dependence Problem	2
1.3	Goal	2
1.4	The Obstacle	3
1.5	Contribution	3
1.6	Thesis Structure	4
2	Background	5
2.1	Capture the Flag and Cryptographic Challenges	5
2.2	Large Language Models in Security	8
2.3	Local Models and Sovereign AI	10
2.4	Symbolic AI and Expert Systems	12
2.5	Neuro-Symbolic AI	13
2.6	The Belief-Desire-Intention Model	13
2.7	Embodied Agents	15
2.8	Retrieval-Augmented Generation	16
3	Related Work	18
3.1	Agentic Approaches to CTF Solving	18
3.2	Evaluating LLMs for Offensive Security	19
3.3	Measuring and Augmenting LLMs for CTF	20
3.4	Sovereign and Local-First AI	21
3.5	Positioning of This Work	22
4	A Neuro-Symbolic Multi-Agent Architecture	23
4.1	Overview	23
4.2	Knowledge Representation	24
4.2.1	Observation Layer	25
4.2.2	CLIPS Expert System	27
4.2.3	RAG Knowledge	28

4.2.4	Constraint Analysis	29
4.2.5	Information Fusion	30
4.3	The Coordinator Agent	31
4.4	The BDI Agent	34
4.5	End-to-End Pipeline Trace	38
4.6	The Explorer Agent	41
4.7	Determinism and Local Execution	44
5	Evaluation	46
5.1	Dataset	46
5.2	Metrics	47
5.3	Experimental Setup	48
5.4	Scenario 1: Raw API Baseline	49
5.5	Scenario 2: External Commercial Agent	50
5.6	Scenario 3: The Proposed System	51
5.7	Results and Discussion	52
5.8	The Role of Each Knowledge Source	54
5.9	Intermediate Experiments	56
6	Conclusion	59
6.1	Summary of Contributions	59
6.2	Limitations	60
6.3	Future Work	61
6.4	Closing	61
	Bibliography	63

Chapter 1

Introduction

This chapter outlines the motivation, scope, and objectives of the thesis. It begins by describing the evolving role of artificial intelligence in automating cybersecurity tasks, before highlighting the limitations and risks associated with relying on large proprietary models. To address these issues, it presents the core goal of the research: developing a neuro-symbolic multi-agent architecture capable of solving security challenges autonomously using small, locally hosted language models. Finally, it details the specific contributions of this work and provides an overview of the document's structure.

1.1 Context

Over the past few years, artificial intelligence has shifted from a research subject to an operational tool in cybersecurity, where it is increasingly used to automate tasks that once required a human expert. In particular, the diffusion of **Large Language Models** (LLMs) has renewed interest in systems capable of reasoning about security problems and of producing working solutions with little human intervention. **Capture the Flag** (CTF) competitions, in which participants solve self-contained security challenges to recover a hidden string known as a *flag*, have become a well-established benchmark for automated reasoning. Each challenge admits a single verifiable answer, the categories cover the whole spectrum of security, and reaching the solution requires both understanding the problem and producing a concrete, executable exploit.

1.2 The Dependence Problem

Most of the recent progress in this area, however, depends on a handful of large proprietary models, reached through commercial Application Programming Interfaces (APIs). This reliance is convenient, but its drawbacks surface as soon as such a system has to run continuously rather than as a one-off experiment. Cost is the immediate one. Running a frontier model is expensive, both in the specialised hardware it requires and in the per-request fees charged by the providers, so that the price of an automated solver grows with the number of attempts it makes, which is exactly the regime in which an iterative, trial-and-error agent operates.

Deeper than cost is the question of availability and control. An API is a service owned by someone else: it can be rate-limited, deprecated, or withdrawn at any time, its terms of use can change overnight, and access can be restricted for commercial or geopolitical reasons, leaving any system built upon it suddenly inoperative. This is the core of the debate on **sovereign AI**, which frames the dependence on a few external providers as a question of strategic autonomy rather than of mere convenience [1]. For a security tool, whose value lies precisely in being available and trustworthy when it is needed, this loss of control is particularly hard to accept.

Most delicate of all is confidentiality. An automated solver for security challenges processes sensitive material: the challenge data itself, and intermediate results that may reveal partial knowledge of a real vulnerability. Routed through a remote API, all of this is transmitted to a third party with every query, leaving the operator's control the moment it is sent. That such offensive AI capability is neither hypothetical nor harmless is by now established, as the first publicly reported cyberattack orchestrated largely by an autonomous AI agent has shown [2]. A tool that wields this capability, therefore, has every reason to keep both the capability and the data it operates on inside a boundary the operator can actually control.

1.3 Goal

This work starts from the opposite premise. Its primary question is architectural: whether the structure placed around a model can matter more than the model's scale, that is, to what extent a small, openly available model, surrounded by explicit domain knowledge, execution, and a feedback loop, can solve security challenges that it cannot solve on its own. Small models are of interest here for a practical reason that goes beyond capability. Unlike a large proprietary model reached through a remote service, a small open-weight model can, in principle be run locally, on hardware controlled by whoever operates it, which removes the

per-request cost, keeps the system available regardless of external commercial or political decisions, keeps sensitive data under the operator’s control, and makes its behaviour easier to reproduce. This kind of operational independence, often discussed under the heading of sovereign AI, is a motivation for the work rather than its central claim; as the later chapters make explicit, the system built here moves towards that ideal without fully reaching it, since one reused component still depends on an external service.

1.4 The Obstacle

The difficulty with this premise is that a small model, taken in isolation, is not powerful enough. When a compact LLM is asked to solve a cryptographic challenge on its own, it usually identifies the right idea but fails to turn it into a correct, runnable program, because it cannot run its own code and therefore has no way to notice and recover from its own mistakes. Execution and tool access do help. Interactive tools have been shown to raise agents’ CTF solve rates substantially while the model is held fixed [3]. However, on their own they are not sufficient for cryptography: in a systematic evaluation, even GPT-4, provided with a code-execution environment, solved none of the cryptographic challenges [4]. What a small local model lacks, therefore, is not only scale but also the structured domain knowledge that turns a recognised weakness into a correct exploit, and it is precisely this knowledge that the architecture in this thesis supplies from the outside.

1.5 Contribution

To bridge this gap, this thesis proposes a **neuro-symbolic multi-agent architecture** that surrounds a small local model with structured domain knowledge and an explicit reasoning process. The system is organised as three cooperating agents: a Coordinator agent that drives the overall flow, a **Belief-Desire-Intention** (BDI) agent that generates, validates, executes, and repairs candidate solutions, and an embodied Explorer agent that runs the code inside an isolated sandbox and reports back what actually happened. The symbolic side, a rule-based expert system together with a retrieval module over past write-ups, supplies the cryptographic knowledge that the model lacks, while the agents turn that knowledge into verified solutions. Agentic approaches to security challenges have been explored before [3]; the distinctive choice here is to obtain comparable behaviour from a small model running locally, by investing in structure rather than in scale.

Concretely, the work is guided by three research questions:

- to what extent a small, locally hosted model, embedded in an appropriate architecture, can solve security challenges that it cannot solve on its own;

- how much of the resulting capability comes from the architecture rather than from the size of the underlying model;
- where the residual failures lie, that is, whether they stem from choosing the wrong technique or from implementing a correct one incorrectly.

In answering these questions, the thesis makes the following contributions:

- **neuro-symbolic multi-agent architecture** for the autonomous resolution of cryptographic CTF challenges, designed to run on local models;
- **symbolic knowledge layer** that combines a CLIPS expert system, a retrieval module, structural constraint filtering, and an adaptive fusion mechanism to rank candidate attacks;
- **embodied execution layer** that runs the generated code inside an isolated sandbox and feeds the real outcome back into the reasoning loop;
- **empirical comparison** across three scenarios, namely a raw model, an external commercial agent, and the proposed system, and across two models, showing that the architecture, rather than the model scale, is what determines the outcome;
- **failure analysis** based on a taxonomy that separates technique-selection errors from implementation errors, which locates the remaining bottleneck in code generation.

1.6 Thesis Structure

The remainder of this document is organised as follows. Chapter 2 introduces the background concepts on which the work builds, from CTF cryptography to language models, symbolic and neuro-symbolic reasoning, the BDI model, and retrieval-augmented generation. Chapter 3 reviews the related work and positions this thesis with respect to it. Chapter 4 describes the proposed architecture in detail, covering the knowledge representation and the three agents. Chapter 5 presents the experimental evaluation across the three scenarios and discusses the results. Chapter 6 draws the conclusions and outlines directions for future work.

Chapter 2

Background

This chapter introduces the foundational concepts and technologies upon which the thesis is built. It first explores the domain of Capture the Flag competitions and the specific nature of cryptographic challenges. Subsequent sections discuss the capabilities and limitations of Large Language Models in cybersecurity, the strategic shift towards local open-weight models, and the principles of both symbolic and neuro-symbolic artificial intelligence. Finally, it outlines the Belief-Desire-Intention reasoning model, the role of embodied agents, and the mechanics of Retrieval-Augmented Generation, providing the necessary theoretical context for the architecture proposed in the following chapters.

2.1 Capture the Flag and Cryptographic Challenges

A **Capture the Flag** (CTF) competition is a security contest in which participants solve a set of self-contained challenges, each hiding a secret known as the *flag*, a string that, once recovered, proves the challenge has been completed. Challenges are conventionally grouped into categories such as web exploitation, reverse engineering, binary exploitation, forensics, and cryptography, and each category exercises a distinct family of skills. Since every challenge has a single, machine-verifiable answer, a CTF provides an objective measure of problem-solving ability, which is also the reason it has become a recurring benchmark for evaluating automated and AI-based solvers [3, 4].

This work concentrates on the **cryptography** category. A cryptographic challenge usually hands the solver a fragment of source code, a set of numeric parameters, or a captured ciphertext, and asks for the original message or a secret key in return. Reaching the flag therefore involves three steps: recognising which scheme is in play, such as the RSA public-key cryptosystem, a stream cipher, or a

block cipher; identifying the specific weakness in the way it has been used; and finally implementing the corresponding computation correctly, often over integers of several hundred digits.

A first practical distinction concerns how the solver reaches the challenge. Some challenges are **remote**: they expose a live service that must be queried over an open network connection, for instance an oracle that decrypts or signs chosen inputs, so that reaching the flag becomes an interactive process. Others are **local**: every value needed is provided up front, and the flag can be recovered from that data without an interactive session against a challenge server. This work concentrates on the local ones, which suit a self-contained, sandboxed execution setting and separate the cryptanalysis and implementation problem from the orchestration of a network session.

Within this category, the challenges considered in this work fall into two structurally different groups, a distinction that recurs throughout the evaluation. **Attack** challenges expose a weakness that must be exploited: a modulus small enough to be factored, two prime factors close enough for Fermat’s method, or an exponent or key reused across messages. Solving them requires knowing the appropriate cryptanalytic technique and applying it. **Construction** challenges, by contrast, involve no adversarial step: every value needed is already available, and the answer follows from a direct computation, such as dividing a modulus by a known factor or recombining two byte sequences through a fixed formula. The two groups stress different abilities, the choice of a correct attack on one side and the faithful implementation of a procedure on the other, and separating them makes the later analysis of failures considerably sharper.

Cryptographic challenges make a demanding testbed for an automated solver precisely because they couple these two abilities. A solver must reason about abstract mathematical structure and, at the same time, produce code that runs correctly on very large numbers, where a single misplaced conversion silently yields the wrong result. The rigid flag format removes any ambiguity about success, so that a solution is either exactly right or simply wrong, with no partial credit. This combination of verifiable outcome, required reasoning, and required implementation is what makes the category well suited to studying where an automated system succeeds and where it breaks down.

Beyond RSA, the evaluation dataset includes two further groups. **XOR stream cipher** challenges provide a ciphertext produced by XOR-ing the plaintext with a keystream derived from a key of known or unknown length. The standard attacks depend on the key reuse pattern: if the key is shorter than the message and repeats, a known-plaintext fragment such as the flag prefix recovers the corresponding key bytes and, by replaying the rotation, the whole keystream. If two messages are encrypted under the same key, their XOR cancels the key and a known-plaintext fragment recovers both plaintexts directly. A small set of **construction** challenges,

drawn from OpenSSL-style exercises, completes the dataset: these are the direct-computation tasks introduced above, here decrypting a ChaCha20 ciphertext with the key and nonce given in the source, recovering a missing RSA parameter by big-integer division, or reconstructing a value through a fixed bitwise formula. Each group requires different recognition patterns and different implementation steps, which in the symbolic knowledge layer means different CLIPS rules with different conditions and different action descriptions.

The structure of a typical RSA challenge illustrates the coupling well. The solver is given a modulus n , a public exponent e , and a ciphertext c , formed as

$$n = p \cdot q, \quad c = m^e \bmod n,$$

and is asked to recover the plaintext m . The textbook scheme recovers it through the private exponent d , obtained from the Carmichael totient $\lambda(n)$:

$$d = e^{-1} \bmod \lambda(n), \quad m = c^d \bmod n.$$

Computing d requires knowing the prime factors p and q , and recovering them from n alone is computationally hard in general, which is the security basis of the scheme. CTF challenges weaken it in specific ways:

- making n small enough for trial division;
- choosing p and q close enough for Fermat’s difference-of-squares method;
- setting $e = 3$ and broadcasting the same message to three recipients, so that Hastad’s broadcast attack [5] applies;
- reusing the same n with two coprime exponents, so that the common-modulus attack recovers the message directly.

Each of these requires recognising the applicable structural pattern and then writing correct arithmetic over numbers of hundreds to thousands of bits, which is exactly the challenge that the architecture in this thesis is designed to meet.

2.2 Large Language Models in Security

A **Large Language Model** (LLM) is a neural network trained on very large amounts of text to predict the next token in a sequence. Once trained, it can be prompted to carry out a wide range of tasks, from translation to code generation, without any task-specific retraining, simply by describing the task in natural language. The same mechanism that makes these models so flexible, the generation of a plausible continuation of the prompt, is also the source of their weaknesses when they are used as autonomous problem solvers rather than as writing aids.

Architecturally, most modern LLMs are based on the **transformer** [6], a neural network design built around a mechanism called **self-attention**. Self-attention allows each position in the input sequence to attend to all other positions, producing a context-aware representation of each token that reflects its relationship to the full input. A transformer stacks many such attention layers, interleaved with feed-forward networks, and the interaction across layers progressively builds more

abstract representations of the input. Text generation is **autoregressive**: the model receives a sequence of tokens, predicts a probability distribution over the vocabulary for the next token, samples from that distribution, appends the sampled token to the sequence, and repeats. The temperature parameter controls the sharpness of the distribution; at temperature 0 the model always selects the highest-probability token, producing a deterministic, greedy output that is maximally conservative.

The capacity of a transformer model scales roughly with its parameter count, which is determined by the depth (number of layers), the width (embedding dimension), and the number of attention heads. A 7B model has substantially more representational capacity than a 1B model, but far less than a 70B model. On programming tasks, this scaling relationship is broadly observed: larger models tend to produce syntactically correct code at a higher rate, to copy literals more faithfully, and to generalise better to algorithms they have not seen in exactly the same form. However, the relationship is not monotonic and not independent of the prompt structure. A well-constructed prompt that supplies the algorithm as a numbered plan before asking for code can raise the performance of a smaller model significantly, because it reduces the number of decisions the model must make autonomously during generation. This observation motivates the chain-of-thought generation step in the BDI agent’s code-generation plan.

The model variant used in this work is an **instruction-tuned** model, meaning that the base pretrained model has undergone a supervised fine-tuning phase on a dataset of instruction-response pairs, in which each example tells the model what task to perform and what a correct response looks like. Instruction tuning shifts the model’s distribution towards helpful, task-following behaviour, making it responsive to natural-language directives that specify what to write rather than requiring carefully engineered prompt formats. Without instruction tuning, a pretrained model treats the prompt as the beginning of a document to continue, which often produces output that is thematically related to the prompt but does not address the specific request. With instruction tuning, the model learns to treat a delimited instruction as a command and the following text as the expected response, which is the behaviour that the BDI agent’s prompt structure relies upon.

In cybersecurity, LLMs have been applied on both the defensive and the offensive side. On the defensive side they support threat detection, log triage, and code review; on the offensive side they are used to explain vulnerabilities, draft exploit code, and drive penetration-testing workflows. Their appeal lies in the breadth of knowledge absorbed during training: a model exposed to large quantities of source code and security write-ups can often recognise a familiar vulnerability pattern and recall the attack that usually defeats it, which is exactly the kind of associative knowledge a cryptographic challenge demands in its first step. This dual-use capability carries a safety cost of its own: even the most capable models

can be coerced past their safety constraints into producing harmful output, the class of attacks known as jailbreaking [7, 8].

This recall, however, does not translate reliably into a working solution, and three limitations recur. First, an LLM cannot execute what it writes: it produces code from its internal distribution alone and never observes whether that code runs or what it prints. Second, it is prone to hallucination, so that it may copy a large number incorrectly, invoke a library function that does not exist, or state a false intermediate result with complete confidence. Third, when it is used in a single pass it has no feedback loop, and once it commits to a wrong approach it has no way to notice the mistake and recover from it. Empirical studies confirm that these limitations cause systematic failures even for capable models on tasks that require precise and verifiable computation [4]. These observations shape the rest of this work: instead of asking a larger model to overcome the limitations on its own, the architecture proposed here supplies the missing execution, verification, and feedback from the outside.

The **code generation** task is worth examining more carefully, because it is more constrained than general text generation. Natural language text is tolerant of small imprecisions: a response that conveys the right meaning with a slightly wrong word still communicates. Code is not tolerant: a missing colon, an incorrect variable name, or a wrong integer literal produces a program that either fails to compile or silently computes the wrong answer. For cryptographic challenges specifically, the integers involved are numbers of 512 to 4096 bits. Copying a 617-digit number incorrectly by even a single character produces a program that runs to completion and prints a result, but the result is not the flag. This class of error is invisible without a comparison against the correct output, which the model does not have access to when it generates the code. These properties, strict correctness requirements and the need for execution feedback, define the requirements for the architecture around the model.

2.3 Local Models and Sovereign AI

The models discussed so far are most often reached as proprietary cloud services. The alternative adopted in this work is to run an **open-weight** model locally, on hardware controlled by whoever operates the system. A local deployment changes both the economics and the guarantees of the system: beyond removing per-request fees and the dependence on an external provider’s availability, it makes the model’s behaviour reproducible, since the same weights on the same hardware can be driven towards deterministic output. These guarantees are what gives substance to the sovereign AI argument: strategic autonomy over a critical system means controlling the models it depends on, rather than renting them [1]. In this work the model and

the reasoning components are deployed this way; one reused retrieval component is an exception that still relies on the OpenAI API, as discussed in Chapters 4 and 5.

Running a capable model on commodity hardware is made possible by **quantisation**, a compression technique that stores each weight with fewer bits, for instance 4 instead of 16, cutting the memory footprint several-fold at a modest cost in accuracy. This work uses a 7B code model in its 4-bit form [9], which fits on a single consumer GPU. The deliberate choice of a small model is central to the thesis: the local and sovereign deployment becomes not merely possible in principle but realistic in practice, since the system can run on an ordinary desktop that many people already own, without the multi-GPU servers, the data-centre allocation, or the rented cloud instances that a frontier model would require. This choice also turns the evaluation into a meaningful test of whether it is the surrounding architecture, rather than the sheer size of the model, that produces a working solver.

The specific model used in this work is **Qwen 2.5-Coder-7B-Instruct** [9], a model from the Qwen family trained specifically for code-related tasks. Its training corpus includes large quantities of code in Python, C, Java, and other languages, as well as technical documentation and code commentary. The instruction-tuned variant used here adds the instruction-following stage described earlier, so it responds directly to natural-language task descriptions. The code specialisation is relevant because the BDI agent requires the model to write correct Python that performs arithmetic over very large integers, a task where a general-purpose language model would underperform a code-specialised one.

The quantisation scheme used in this work is **NF4** (4-bit Normal Float) [10], a data type designed for weights drawn from a normally distributed population. Standard 4-bit integer quantisation rounds each weight to one of 16 uniformly spaced levels, which distributes the representational budget evenly over the weight range. NF4 instead places the 16 quantisation levels at positions that minimise the expected quantisation error for a normal distribution, concentrating levels near zero, where most weights cluster, and spacing them more widely in the tails where weights are rare. The result is a lower average quantisation error at the same bit width. Practically, NF4 stores each weight in 4 bits instead of 16, shrinking the weight footprint of a 7B model to roughly a quarter of its full-precision size, about 4 gigabytes, so that it fits on a single mid-range consumer GPU without any change to the model architecture or the training procedure.

2.4 Symbolic AI and Expert Systems

Before the current dominance of statistical learning, artificial intelligence was largely **symbolic**: knowledge was represented explicitly, as facts and rules, and conclusions were obtained by manipulating those symbols according to rules of inference. An **expert system** is a direct application of this idea, encoding the knowledge of a human specialist as a collection of *if-then* rules and applying them to a set of facts in order to draw conclusions within a narrow domain. Its defining virtue is transparency: every conclusion can be traced back to the rules that produced it.

A common inference strategy for such systems is **forward chaining**, in which the engine starts from the known facts and repeatedly fires every rule whose conditions are satisfied, asserting new facts until no further rule applies. **CLIPS** (C Language Integrated Production System), a rule-based language created for building expert systems, implements exactly this strategy and is the engine used in this work. Its rules are declarative and independent of one another, so that domain knowledge can be extended by adding rules rather than by editing control flow, and each rule carries a *salience* value through which the designer orders competing rules by priority. These properties make the knowledge base easy to inspect, to audit, and to grow, which is precisely what is needed for a body of cryptographic expertise that must remain trustworthy.

CLIPS implements the **Rete algorithm** [11], a data-flow network that matches the conditions of all rules simultaneously against the current fact base without re-evaluating every rule from scratch at each cycle. Nodes in the network correspond to the individual condition elements of rules, and tokens propagate through the network as facts are asserted or retracted. Only the parts of the network affected by a change are re-evaluated, which makes forward chaining efficient even when the rule base is large. For the cryptographic knowledge base in this work, the Rete optimisation is less critical because the fact base for any single challenge is small, but the algorithm’s correctness guarantee, that no rule activation is missed and no rule fires on stale facts, is valuable in a setting where a wrong suggestion would waste a full BDI attempt.

The role of expert systems in AI has fluctuated significantly since their peak in the 1980s. They fell out of favour as machine learning made it possible to build classifiers and regression models from data rather than from hand-coded rules, and as the brittleness of early rule bases became apparent in domains too complex to fully formalise. The resurgence of interest in combining symbolic and neural approaches reflects a recognition that each method has genuine strengths in different conditions. For a domain such as cryptographic attack selection, where the applicable techniques are a finite, enumerable set, where the conditions for each technique are precise and mathematical, and where a wrong selection is costly in compute time, an explicit rule base is better suited than a classifier that can only

be inspected through its outputs. This is the sense in which the symbolic layer in this work is not a legacy component but an architectural choice.

2.5 Neuro-Symbolic AI

Symbolic and neural approaches have complementary profiles. Symbolic systems are precise, transparent, and verifiable, but brittle: they only know what has been written into them and cope poorly with inputs that do not fit their rules. Neural models, on the contrary, are flexible and tolerant of noise, drawing on a vast implicit knowledge, yet they are opaque and produce confident errors that are hard to detect. **Neuro-symbolic AI** is the line of research that combines the two families so that each is used where it is strongest [12].

The system presented in this thesis is neuro-symbolic by design, with an explicit division of labour. The neural component, the LLM, is used for what it does well: recalling common patterns and turning a described procedure into code. The symbolic component, the expert system together with the structured knowledge built around it, supplies precise and verifiable domain knowledge and keeps the strategic decisions, namely which attack to attempt and in which order, deterministic. In this way the unreliable, generative part of the system is confined to writing code, while the choice of what to compute remains under symbolic, inspectable control.

Kautz [12] surveys six designs for such integration, ranging from a neural pattern-recognition subroutine embedded within a symbolic problem solver (his *Symbolic[Neuro]* design, exemplified by AlphaGo) to a symbolic reasoning engine embedded inside a neural one (his *Neuro[Symbolic]* design). Most of these designs couple the two paradigms tightly, within a single reasoning process, whether by embedding one in the other or by pipelining them. The integration in this thesis is deliberately more modular: the CLIPS engine and the LLM remain separate components that do not share internal representations, and they communicate only through structured strings that represent suggested techniques and generated code. This loose coupling makes each component independently replaceable and independently testable, which is a practical advantage in a research context where both the knowledge base and the underlying model are likely to evolve.

2.6 The Belief-Desire-Intention Model

The **Belief-Desire-Intention** (BDI) model is an architecture for rational agents that grows out of a philosophical account of practical reasoning [13], describing how an agent decides what to do on the basis of what it believes and what it wants. It was turned into a concrete and influential agent architecture by Rao and Georgeff [14], and it organises an agent’s state around three attitudes. **Beliefs**

represent what the agent currently holds to be true about the world; **desires** are the states of affairs it would like to achieve; and **intentions** are the desires it has actually committed to pursuing, each bound to a plan that should bring it about.

On this basis the agent runs a continuous cycle: it perceives the environment and revises its beliefs, generates the desires whose conditions now hold, deliberates to decide which of them to commit to, selects a plan for the chosen intention, and executes the next step of that plan, repeating until the goal is reached or abandoned. This explicit separation of what the agent knows, wants, and is committed to is valuable for an agent that must generate and repair code: the state of the reasoning is laid out in inspectable beliefs, priorities can be expressed by ordering desires, and a plan that fails can be replaced without discarding the underlying goal. These are the reasons the model was chosen as the reasoning core of the system, and the concrete adaptations made to it are described in Chapter 4.

A key notion in BDI design is the **commitment strategy**, which governs when an agent may abandon an intention it has already adopted. The classic discussion distinguishes three positions by when the agent is willing to abandon an intention. A *blind-committed* agent pursues an intention until it believes the goal has been achieved, ignoring all other changes in its beliefs; this is the most persistent but also the most inflexible posture. A *single-minded* agent is prepared to abandon it sooner: it maintains the intention until it believes the goal has been achieved, or until it comes to believe that the goal is no longer achievable. An *open-minded* agent is the most flexible of the three, holding on to an intention only as long as it still believes the goal achievable and still desires it, so that a change in its beliefs or its goals is enough to prompt reconsideration. The concrete BDI agent in this work sits close to the open-minded end: at every cycle it checks whether its current desires can still be satisfied and whether its current plan should be replaced, with the cycle boundary providing a natural reconsideration point rather than a continuous re-evaluation. The fixed cycle budget of 30 iterations imposes a hard commitment horizon, after which the agent terminates regardless of remaining beliefs, handing control back to the Coordinator agent. This budget is not arbitrary. In practice, a challenge that the agent is able to solve is solved within a few cycles. When the agent is still failing once the horizon is reached, it is usually trapped in an unproductive repair loop, regenerating the same ineffective fix at every cycle, and further iterations would not converge on a solution. Cutting the attempt off therefore costs no realistic chance of success, and it frees the budget for the next hypothesis.

A further property of BDI systems that is relevant here is the distinction between **deliberation** and **means-ends reasoning**. Deliberation selects which desire to commit to from among those currently active, using priority to resolve conflicts when multiple desires are simultaneously applicable. Means-ends reasoning selects, from the plan library, a concrete plan for the chosen desire, given the current beliefs.

This two-level structure allows the architecture to express both strategic priorities, for example that fixing a syntax error is more urgent than generating new code, and tactical variety, for example that there may be multiple repair strategies to try in sequence. The explicit separation of these two reasoning steps, which in a simple loop are collapsed into a single function call, is the primary reason the BDI model was adopted over a simpler iterative structure.

2.7 Embodied Agents

An **embodied agent** is an agent whose cognitive processes are coupled to an environment through direct perception and action. The contrast with a purely symbolic reasoner is fundamental: instead of operating over an internal world model built before execution begins, an embodied agent observes the actual state of the environment and modifies it through actions, so its beliefs are grounded in what the environment reports rather than in what it predicted.

The resulting **perceive-act loop** is simple in structure: the agent reads its current observations, selects an action, applies it, and receives updated observations in return. The value of this design is that the agent does not need to anticipate every possible outcome in advance. It can form a hypothesis, act on it, and let the environment answer whether the hypothesis was correct. Errors become self-revealing: a wrong action produces an unexpected observation, and that discrepancy is what triggers revision.

In a computational setting this principle translates into an agent that does not merely generate artefacts but actually executes them and observes the result. An agent that runs code inside a real environment receives genuine feedback, not a prediction: the output of a computation, the content of a file, or the traceback of an exception. This feedback loop is what gives the agent the ability to detect and correct implementation errors that would be invisible to a model reasoning purely in text. The role this plays in the architecture of this work, and the isolated environment inside which it operates, are described in Chapter 4.

The choice of environment matters because it defines what can and cannot be observed, and what actions are available. An uncontrolled environment allows the agent arbitrary access to the host system, which raises safety and reproducibility concerns: a solver that reads from network sockets, modifies system files, or consumes unbounded resources is difficult to reason about and dangerous to run at scale. In this work the environment is a Docker container configured with strict isolation: a read-only root filesystem, a limited writable workspace, a capped process count, a bounded memory allocation, and a restricted network policy that allows only specific outbound connections. This constrained environment is both safer and, paradoxically, more predictable for the agent, because the set of possible

observations is smaller and the consequences of each action are more local. The sandboxed environment is also a form of **grounding**: the agent’s belief that its code executed correctly is not a linguistic inference but a direct observation of the exit code and standard output of a real process. This grounding in execution is what distinguishes the system from an agent that reasons entirely in text and hopes that its reasoning corresponds to a real computation.

2.8 Retrieval-Augmented Generation

A language model stores knowledge implicitly in its weights, as patterns absorbed during training. That parametric knowledge has two inherent limits: it is bounded by the training corpus, and it is frozen at the training cutoff. **Retrieval-Augmented Generation** (RAG) [15] addresses both limits by supplementing the model at inference time with documents retrieved from an external collection. Given a query, the system identifies the most relevant documents in the collection and appends them to the model’s context, so the model can condition its output on both its parametric knowledge and the retrieved material.

For a small, locally deployed model the benefit is concrete. A 7B model cannot be expected to have absorbed every write-up from every past security competition during training. A RAG component makes that accumulated knowledge available without retraining the model and allows the knowledge base to grow simply by adding new documents.

In this work the collection consists of CTF write-ups, post-challenge explanations written by participants who solved comparable problems. When a new challenge arrives, its description is used to query a vector index over the collection; the closest matches, selected by semantic similarity of their embeddings, are injected into the context alongside the problem statement. The intent is to surface the specific cryptanalytic approaches that have worked on similar challenges before, giving the local model access to concrete precedents rather than requiring it to reconstruct the correct attack from first principles alone. The implementation of this component is detailed in Chapter 4.

The vector index works by representing each document as a dense numerical vector, an **embedding**, produced by a model trained to place semantically similar texts near one another in the embedding space. At query time the query text is embedded by the same model, and the documents whose embeddings are closest to the query embedding, typically by cosine similarity, are retrieved. Cosine similarity is the standard choice over Euclidean distance because it is invariant to the magnitude of the embedding vectors, measuring only the angle between them; this makes it more robust to document length variation, since a longer write-up and a shorter one that describe the same attack will be oriented similarly in the

embedding space regardless of their different norms.

For the specific application of CTF write-ups, this semantic retrieval has an advantage over keyword search: write-ups for the same technique may use different terminology, describing the same attack under the names Fermat factorisation, close-prime factorisation, and factor by proximity, none of which necessarily appear in the challenge text. An embedding model trained on code and security text will place these documents close to one another and close to a query that describes the challenge without using any of those terms. The retrieval therefore functions as a form of technique identification in its own right, operating independently of the CLIPS rules and providing a complementary signal to the expert system.

Chapter 3

Related Work

The automated solving of security challenges has attracted growing interest alongside the general advance of large language models in code and reasoning tasks. The works most relevant to the present project address three overlapping questions: how far LLMs go when equipped with interactive tools, how hybrid approaches combining retrieval and execution change the picture, and what the case is for deploying these systems without depending on large proprietary providers. This chapter surveys the most pertinent contributions and identifies, in the final section, where the system proposed here departs from them.

3.1 Agentic Approaches to CTF Solving

The most direct precursor to this work is **EnIGMA** [3], a single LLM agent designed to solve CTF challenges across all categories. EnIGMA frames the problem as an agent-computer interaction task: the LLM agent is given a set of interactive tools, a bash shell, decompilers, and debuggers, and is instructed to use them in sequence to explore the challenge and capture the flag. Its central finding is that these interactive tools substantially raise solve rates, achieving state-of-the-art results across several CTF benchmarks: access to a real execution environment changes what is achievable without any change to the underlying model.

EnIGMA is built on top of the SWE-agent [16] **agent-computer interface** (ACI) paradigm, which filters and formats observations so the model can act more reliably on a simplified view of the machine. The architecture is deliberately general: the same scaffolding is applied across challenge categories without any category-specific knowledge. The models used are leading systems, GPT-4 Turbo, GPT-4o, Claude 3.5 Sonnet, and LLaMA 3.1 405B, so the architectural contribution is demonstrated against a backdrop of very large frontier and open models. This work asks the same structural question, whether the architecture is the decisive

factor, but from the opposite starting point: a 7B local model.

The general-purpose nature of the ACI paradigm is both its strength and its limitation from the perspective of this work. Because the agent receives raw tool outputs and must interpret them without any domain-specific framing, the quality of its decisions is bounded by what the underlying model can infer from unstructured text. For cryptographic challenges, where the correct technique is determined by precise mathematical conditions on the parameters, unstructured tool output is an inefficient representation: the agent must re-derive the relevant conditions from scratch on each attempt rather than matching them against a pre-computed domain model. The explicit knowledge layer in the system proposed here is the architectural response to this inefficiency.

3.2 Evaluating LLMs for Offensive Security

A systematic evaluation of LLMs on CTF challenges is provided by Shao and colleagues [4], who test six models on a broad set of challenges under both a fully-automated workflow and a human-in-the-loop one. Their headline result is that capable models are already strong: GPT-4 outperforms most human participants overall. Two findings, however, are directly relevant here. First, cryptography is a wall: GPT-4, even with access to a code-execution environment and standard tools, solves none of the cryptographic challenges, falling below the human success rate for that category. Second, the fully-automated setting, in which the model receives no feedback, performs markedly worse than the human-in-the-loop one, and the authors report that supplying feedback substantially reduces failures. Together these point to the gap this work targets: for cryptography, neither raw capability nor bare tool access is sufficient, and a model used in a single pass, without a feedback loop or domain structure, fails systematically.

The system proposed here is designed against exactly this gap. Execution is handled by the Explorer agent inside a real Docker sandbox; hallucinated values or broken code are caught by running the script and reading the output; the failure classifier together with the hypothesis-cycling mechanism provide the feedback loop that a single-pass LLM call lacks; and the symbolic knowledge layer supplies the cryptographic domain structure that tool access alone does not.

This work further adopts a two-way decomposition of failures that its own analysis makes precise: the distinction between a *technique-selection error*, in which the system attempts the wrong attack, and an *implementation error*, in which the system selects the correct attack but fails to implement it. The two types of failure have different root causes and different remedies. Technique-selection errors are addressed by the symbolic knowledge layer, which encodes the conditions under which each attack applies. Implementation errors are addressed by the BDI

repair loop, which detects them through execution feedback and attempts to fix the code rather than abandoning the approach. The failure taxonomy in Chapter 5 follows this decomposition directly and uses it to assess where the remaining failures concentrate.

3.3 Measuring and Augmenting LLMs for CTF

Ji and colleagues [17] approach the problem from two directions simultaneously. They first construct **CTFKnow**, a benchmark of nearly four thousand questions extracted from over a thousand write-ups across major competitions, designed to measure LLMs’ technical knowledge of CTF scenarios separately from their reasoning and execution ability. The benchmark reveals a consistent pattern: LLMs perform well on single-choice questions that test parametric knowledge in isolation, suggesting that the training corpus covers most common attack patterns, but the same knowledge does not transfer reliably to open-ended questions that require matching the correct technique to a specific scenario. The gap between what a model knows and what it can deploy is one of the central observations of the work.

That gap motivates the augmentation layer they propose, named **CTFAgent**, which introduces two modules. The first is a two-stage retrieval-augmented generation component that injects relevant retrieved knowledge at both the understanding and the exploitation stage of challenge solving. The second is an interactive environmental augmentation component that gives the model access to command-line tools during execution. On two widely used CTF datasets the combined system achieves improvements of over 80% relative to the baseline, and it ranked in the top quarter of a major public competition with nearly seven thousand participating teams.

CTFAgent is the closest existing work in spirit to the system proposed here: it combines retrieval and execution, and it recognises that knowledge injection matters as much as the model itself. The key differences are in the design of the knowledge layer and in the deployment model. CTFAgent uses a generic RAG component without structured domain knowledge: the model is given relevant text and expected to extract and apply the appropriate technique on its own. The system developed in this work replaces the generic retrieval with an expert system that encodes cryptographic attack knowledge as explicit, salience-ordered rules; the RAG component operates alongside the expert system rather than as the sole channel. The second difference is in the model and its deployment: CTFAgent relies throughout on a large proprietary model, whereas the reasoning core here runs on a small local open-weight model, and only the reused retrieval component depends on an external service. This contrast between local control and model scale is taken up in the next section.

3.4 Sovereign and Local-First AI

The works reviewed above share a common assumption: the underlying LLM is a frontier model accessed through a commercial API. Singh and Sengupta [1] articulate why that assumption deserves scrutiny. Their analysis, framed as an argument for sovereign AI, examines the geopolitical and operational exposure created by routing critical computational tasks through a small number of providers whose availability is neither guaranteed nor politically neutral. The argument has direct implications for automated security tooling: a system whose reasoning depends on an external service is exposed to that service being throttled, repriced, or unavailable at the moment it is needed, and it transmits potentially sensitive problem data to a third party with every query.

The case for local deployment is therefore not primarily a question of cost but of control, and of keeping sensitive information within a boundary the operator governs. The system proposed here keeps its reasoning core local: the model, the expert system, the constraint checking, and the code generation all run on local hardware. It approaches operational independence for the whole pipeline except one component, the retrieval module, which is reused from a previous thesis as a compiled executable and calls the OpenAI API internally; this component could not be reconfigured to a local model within the scope of the work. The primary contribution is therefore the architectural claim that the surrounding structure compensates for the difference in model scale, and operational independence is a secondary aim that the work approaches rather than fully attains.

Singh and Sengupta frame the sovereign AI argument primarily at a national or institutional level, noting the strategic risk of building critical infrastructure on technology owned and operated by foreign companies. Their own conclusion is in fact a balanced one: sovereignty in AI calls for managed interdependence rather than outright isolation. What this work borrows from it is the narrower concern that the analysis raises, namely the strategic exposure of routing a critical tool through a provider that one does not control. The same argument applies at the level of an individual security team or research group: an automated solver that processes challenge data, and potentially intermediate results that might reveal partial knowledge of a vulnerability, should ideally not transmit that data to an external service as a routine part of its operation. The system here honours this principle for its reasoning core but not yet for the retrieval module, which is the external dependency noted above and a target for future work. The Docker sandbox, by contrast, extends the principle inward for the executed code: the solver process itself is isolated from the host network, so that even if the generated code attempted exfiltration, the egress policy would block it.

3.5 Positioning of This Work

The related work converges on a consistent finding: augmenting an LLM with execution access, and with retrieved knowledge, substantially raises its capability on CTF tasks. EnIGMA demonstrates the effect of execution access, CTFAgent that of retrieval and execution combined, and the systematic evaluation of Shao and colleagues explains why the unaugmented baseline fails. The system proposed here accepts the same premise but pursues it along two axes that none of the reviewed works address together.

The first axis is the representation of domain knowledge. EnIGMA and CTFAgent supply knowledge implicitly, through retrieved text or generic tool access. The system described here makes domain knowledge explicit, encoding it as logical rules in a CLIPS expert system with confidence scores and salience-ordered priorities. This makes the knowledge base inspectable, auditable, and extendable independently of the model, properties that matter in a security context where the reasoning behind a decision is as important as the decision itself.

The second axis is the deployment model. Every CTF-solving system reviewed above uses a large frontier model as its backbone, which limits the practical independence of the resulting system. The evaluation in Chapter 5 tests whether a 7B local model, embedded in the proposed architecture, can hold up against both a raw single-prompt baseline and a commercial agentic baseline, each run with a stronger commercial model alongside the local one. The result is the central empirical claim of the thesis.

Taken together, these two axes define a position in the design space that none of the reviewed works occupies: explicit domain knowledge combined with local deployment. The choice is driven less by performance than by the principle that a system used in a security context should be inspectable and independent. The explicit knowledge layer allows the system’s decisions to be audited rule by rule; running the reasoning core on a local model removes, for that core, the dependency on external providers whose availability and trustworthiness cannot be verified, and the one component that still relies on such a provider, the reused retrieval module, is discussed as a limitation in Chapter 5. The evaluation in the following chapter demonstrates that this position does not require sacrificing solve rate: even on the local 7B model the proposed system outperforms the commercial agentic baseline run on the stronger model, and it closes much of the distance to the same architecture driven by that stronger model.

Chapter 4

A Neuro-Symbolic Multi-Agent Architecture

This chapter presents the design and implementation of the proposed architecture. It details the system’s core components, starting with a high-level overview of the end-to-end solving pipeline. Subsequent sections describe the knowledge representation layer, the orchestration driven by the Coordinator agent, the cognitive reasoning loop of the BDI agent, and the embodied execution environment managed by the Explorer agent. Finally, it discusses the role of determinism and the operational benefits of local execution.

4.1 Overview

The system takes as input a directory of CTF challenges, each stored as a text file containing the problem description, source code, and numerical parameters, and processes them one at a time, attempting to produce the correct flag for each as output. It is organised as a pipeline of three cooperating agents, each responsible for a distinct part of the work, coordinated by a central orchestrator.

At the top level, the **Coordinator agent** (implemented as **FirstAgent**) drives the overall solving flow. It first attempts a fast-path direct solve, which succeeds only for the rare challenge whose answer can be read off the input without any computation. Every other challenge falls through to the full pipeline: the Coordinator agent gathers structured knowledge from multiple sources, ranks the resulting hypotheses, and then hands each one in turn to the BDI agent.

The **BDI agent** (**TrueBDIAgent**) receives a hypothesis in the form of a natural-language description of the cryptographic technique to apply and runs a multi-step cognitive cycle to turn that description into a working Python solver. It generates code, statically validates it, sends it to the Explorer agent for execution, reads the

result back, and, if the code fails, diagnoses the error and repairs it. This loop continues for up to 30 cycles or 15 minutes, whichever comes first.

The **Explorer agent** (`SystemOperatorAgent`) is the embodied component. Its only job is to execute commands and manage files inside a Docker sandbox. It perceives the container filesystem, plans the required action by calling its own LLM, checks the action against a safety guardrail, and dispatches it through a thin bridge (`DockerExecutor`) that communicates with the running container via `docker exec`.

The end-to-end flow, summarised in Figure 4.1, proceeds as follows:

1. The Coordinator agent reads the challenge text and calls the LLM for a preliminary vulnerability extraction.
2. The observation layer, the RAG server, and the CLIPS expert system run over the challenge text and produce their fragments of knowledge; the constraint analyser then penalises the expert suggestions that are structurally impossible for the challenge, demoting them in the ranking.
3. The information fuser merges the retrieved knowledge and the surviving expert suggestions, weighted by the observation tags, into a ranked list of hypotheses.
4. For each hypothesis the Coordinator agent launches a fresh BDI agent, which generates and executes a solver script through the Explorer agent inside the Docker sandbox.
5. If the solver succeeds, the flag is extracted and verified. If it fails, the failure classifier decides whether to retry the same hypothesis with a fix hint or to advance to the next one.
6. After every attempt the outcome is recorded in the model’s persistent history, so future challenges of the same type start from a better-informed ranking.

The following sections describe each layer of this pipeline in detail.

Figure 4.2 traces the messages exchanged between the three agents during a single solve attempt that requires one repair cycle.

4.2 Knowledge Representation

Before the BDI agent generates any code, the system gathers and refines knowledge about the challenge through four components: the observation layer, the CLIPS expert system, the RAG module, and the constraint analyser. The results are combined by a fifth component, the information fuser, into a single ranked list of hypotheses. Figure 4.3 shows these components and their fusion in detail.

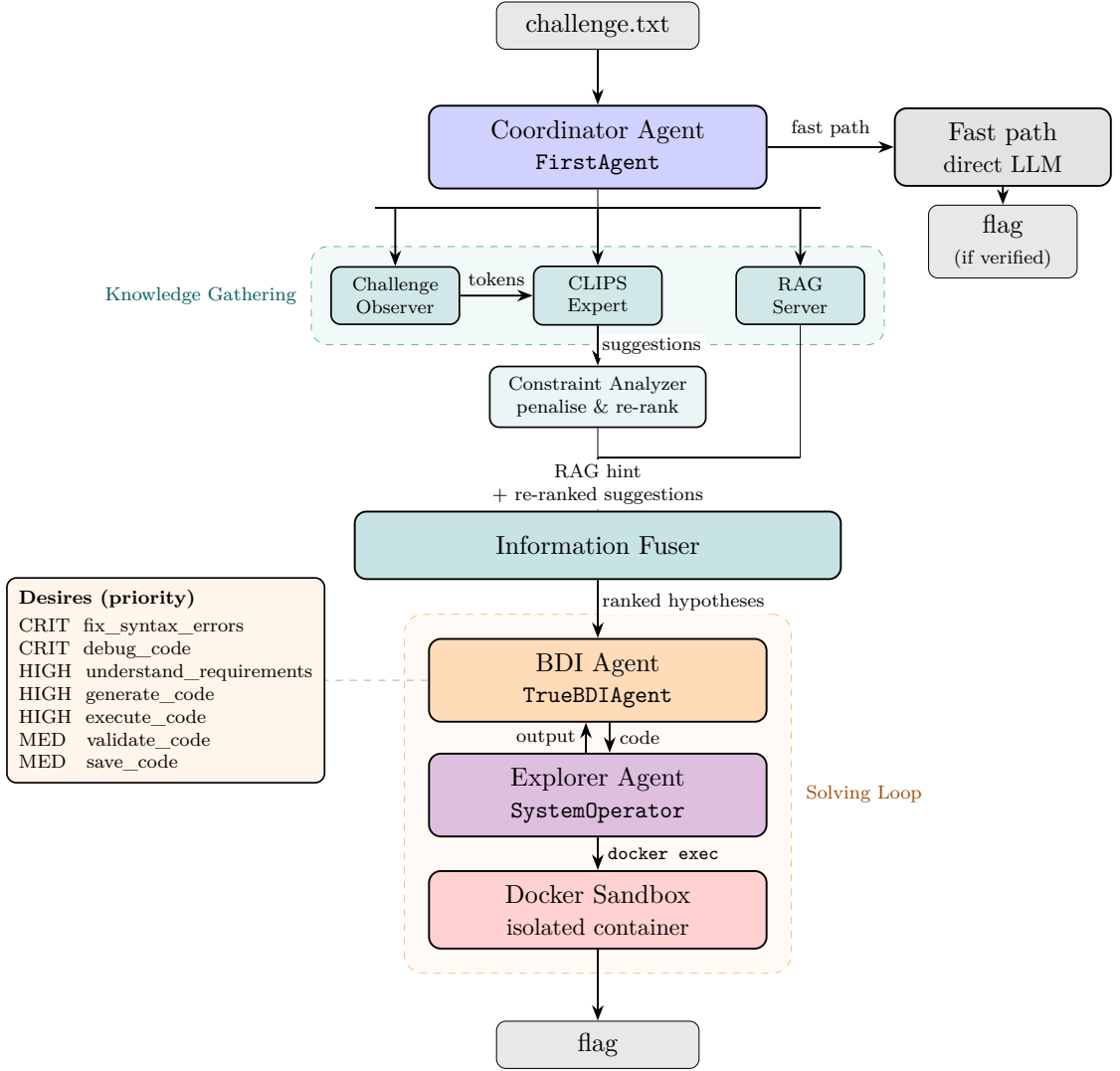


Figure 4.1: Complete architecture of the proposed system. The Coordinator agent attempts a fast path first, then runs its knowledge-gathering stage: the observation layer supplies structured tokens to the CLIPS expert system, while the RAG server retrieves relevant write-ups. The constraint analyser sits on the expert branch, penalising and re-ranking the CLIPS suggestions that are structurally impossible before fusion. The Information Fuser merges the RAG hint and the re-ranked expert suggestions into a ranked hypothesis list. The BDI agent generates and repairs solver code inside a loop; the Explorer agent executes it in an isolated Docker sandbox.

4.2.1 Observation Layer

The observation layer (**ChallengeObserver**) is the first stage of every run. It performs two deterministic regex passes over the raw challenge text and produces two output structures.

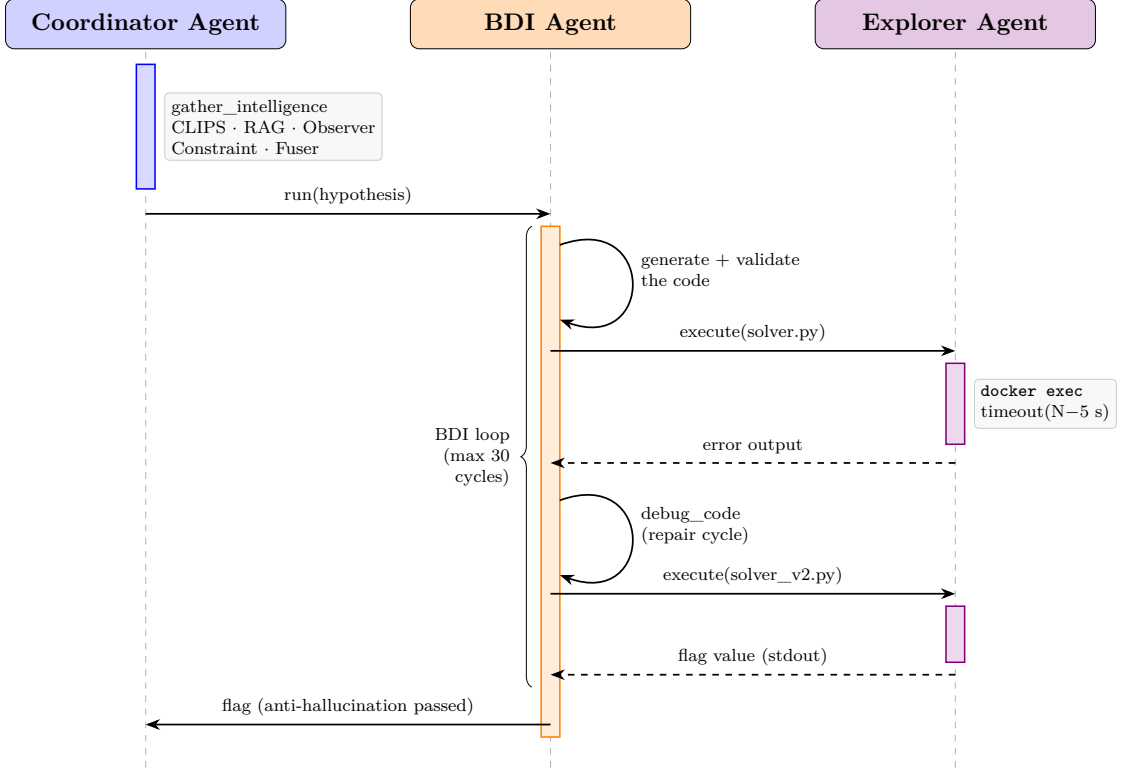


Figure 4.2: Sequence of inter-agent messages during one solve attempt that requires a single repair cycle. The Coordinator agent gathers knowledge and launches the BDI agent with the top-ranked hypothesis. The BDI agent first generates and validates the solver internally, and only then has the Explorer agent execute it in the sandbox; the error output triggers one `debug_code` cycle before the corrected code succeeds.

The first output is a list of **observation tags**: short category strings such as "rsa", "xor", "aes", "ecc", "prng", "hash", or "lattice", obtained by matching word-boundary patterns against the challenge text. The patterns are written with care to avoid false positives: for instance, the tag "dh" requires the word *diffie* or *discrete_log*, not just any occurrence of the letters *d* and *h*. These tags are not used for CLIPS; they feed the Jaccard similarity scorer in the fusion history, which uses them to measure how closely a past challenge resembles the current one.

The second output is a string of **observation tokens**: structured signals of the form `[OBS:...]` injected directly into the text that the CLIPS engine reads. The tokens are extracted partly by regex on the raw challenge text (for example, `[OBS:E-3]` when the exponent *e* is found equal to 3, or `[OBS:CLOSE-PRIMES]` when the source code contains `nextprime` or `close primes`) and partly by mapping the vulnerability names returned by the LLM's preliminary extraction step through a fixed dictionary. The rationale is that CLIPS rules match substring patterns in an

uppercase version of the challenge text, so raw keywords can produce false positives when challenge filenames or flavour text happen to contain them. The observation tokens restate the same information in an unambiguous, challenge-name- and prose-phrase-independent form.

The entire observation layer involves no LLM calls and produces identical output for identical input. This is intentional: the observation step is the one part of the pipeline that must be fully reproducible across runs.

4.2.2 CLIPS Expert System

The expert system is the symbolic core of the architecture. It is implemented in CLIPS (C Language Integrated Production System) using a knowledge base of declarative if-then rules stored in `ctf_kb3.clp`. The Python wrapper `CTFInferenceEngine` initialises a fresh CLIPS environment for each challenge, loads the rule file, asserts a single fact containing the uppercased challenge text concatenated with the observation tokens, and runs the forward-chaining engine until no further rule can fire.

Each rule in the knowledge base has the following structure. A **salience** value (an integer priority; the attack rules in this base sit in the range of roughly 50 to 97) sets its order relative to competing rules: when two rules can fire simultaneously, the one with higher salience fires first. A set of **conditions** specifies the keyword patterns that must be present in the input text for the rule to activate. A **guard** prevents the same suggestion from being asserted twice. When all conditions are satisfied, the rule’s consequent asserts a **suggestion fact** containing seven fields: a unique integer **id**, the broad category (for example **rsa**), the specific subcategory and attack name (**rsa-fermat**), a natural-language action description giving the algorithm as pseudocode steps for the code-generation prompt, a tool hint naming the key library calls, a confidence integer (0–100) expressing how reliably this technique applies when the pattern is present, and a reason string explaining the mathematical justification.

The knowledge base spans 21 cryptographic categories across 99 rules, with specific attack rules for RSA factorisation attacks (small- n factorisation, Fermat’s method, Hastad broadcast, common modulus, Franklin-Reiter, Boneh-Durfee), AES modes (CBC padding oracle, ECB block reuse, GCM nonce reuse), XOR stream ciphers (direct known-plaintext attack, rotating key, two-time pad), elliptic curve cryptography (ECDSA nonce reuse, Smart’s attack, Pohlig-Hellman), discrete logarithm problems, PRNG state recovery (Mersenne Twister), hash-length extension, and classical substitution ciphers.

As a concrete illustration, the rule for Fermat’s factorisation method is reproduced verbatim from `ctf_kb3.clp` below (the long **action** and **reason** strings are single logical strings, wrapped here only to fit the page). The rule fires at salience 85

when the input text carries an RSA marker (such as RSA or .PEM) together with a marker of near-equal primes (such as FERMAT, NEXTPRIME, or CONSECUTIVE PRIME), and when a suggestion for this technique has not already been asserted:

ctf_kb3.clp

```

1 (defrule rsa-fermat
2   (declare (salience 85))
3   (input-text (body ?t))
4   (test (any ?t "RSA" ".PEM"))
5   (test (any ?t "FERMAT" "P AND Q CLOSE" "FERMAT FACTORIZATION"
6               "NEXTPRIME" "NEXT_PRIME" "CONSECUTIVE PRIME"))
7   (not (suggestion (subcategory rsa-fermat))))
8 =>
9   (assert (suggestion
10    (id (next-id)) (category rsa) (subcategory rsa-fermat)
11    (action "Fermat Factorization: a=isqrt(n)+1; increment a until
12           b^2=a^2-n is a perfect square. p=a-b, q=a+b. If
13           nextprime(p) was used, gmpy2.isqrt converges in <10 steps.
14           ")
15    (tool "gmpy2.isqrt / gmpy2.iroot")
16    (confidence 92)
17    (reason "q=nextprime(p) means p,q differ by -log(p) bits => n is
18           near a perfect square => Fermat trivially fast")
19  ))

```

The confidence of 92 reflects that the Fermat method is highly reliable when the precondition holds, but it is not the maximum because the trigger tokens may occasionally appear without the actual primes being close enough for the method to terminate within the sandbox timeout.

The principal advantage of encoding this knowledge in CLIPS rather than as Python conditional logic is separation of concerns: adding a new attack requires appending a single rule to the .clp file without modifying any Python code, the salience attribute manages priority automatically, and the rule body is readable by anyone familiar with the domain without knowledge of the surrounding code. The output of the engine is a list of **Suggestion** objects sorted by confidence, which is then passed to the constraint filter.

4.2.3 RAG Knowledge

Alongside the expert system, the Coordinator agent queries a **Retrieval-Augmented Generation** server, reused from a previous thesis, that runs locally as a Gradio application. The server maintains a vector index built from a collection of past CTF write-ups. For each challenge, the Coordinator agent sends a payload

containing the challenge name, the full challenge text, and the list of vulnerabilities extracted by the preliminary LLM call. The server embeds the query, retrieves the write-ups most similar to it, and uses a language model to generate a structured response conditioned on the retrieved passages. This component is a compiled executable and is therefore a black box from the perspective of this work: its language model is served through the OpenAI API rather than by the local model, and its internal retrieval and generation parameters cannot be changed. It is the one part of the pipeline that is not fully local, a point taken up in the discussion of operational independence in Chapter 5.

The query uses a two-step protocol. The first exchange asks the server to identify the relevant write-ups and extract a high-level description of the technique they describe. The second exchange imposes a strict output format: exactly two lines, one giving the sub-category in the form `topic = "crypto/<subcategory>"` and one giving a natural-language procedure in the form `requirements = "Generate Python solver: ... steps: 1. ... N. print flag"`. The Coordinator agent parses only these two lines, discarding any surrounding text.

The RAG component compensates for limitations in the expert system: if a challenge type is not covered by any CLIPS rule, a write-up from a sufficiently similar past challenge may still provide the relevant technique. However, when the vector index contains no close matches, the server returns a generic response shorter than 80 characters, which is penalised heavily by the quality evaluator in the fusion step. In this case the RAG hypothesis is effectively demoted and the expert system takes precedence.

4.2.4 Constraint Analysis

CLIPS rules match text patterns without analysing the mathematical structure of the challenge. Some attacks require conditions that can only be determined by inspecting the actual values and the code structure: Hastad’s broadcast attack requires at least 3 independent ciphertext-modulus pairs, common-modulus attacks require at least 2 ciphertexts, Mersenne Twister state recovery requires at least 624 outputs, and so on. Proposing these attacks when the necessary conditions are not met wastes a full BDI attempt.

The **ConstraintAnalyzer** addresses this by extracting a **ChallengeConstraints** object from the challenge text before the suggestions reach the hypothesis ranker, through three stages that are then merged. First, a deterministic regex pass fixes the cryptographic type and the key size in bits and provides initial counts of recipients and ciphertexts. A targeted LLM call then returns structural counts, the number of independent targets and the number of total outputs, which are merged with the regex counts by keeping the larger value. Finally, a second LLM call captures three semantic flags that depend on the intent of the code rather

than its surface form: whether the key or nonce is static across messages, whether a known-plaintext fragment such as the flag prefix is available, and whether a nonce is actually reused rather than merely initialised to a fixed value. Both calls run on the Coordinator’s configured language model, the same one that performs the preliminary extraction: the local Qwen model in the local configuration and GPT-4o-mini in the commercial one. When an LLM call fails, a keyword fallback provides a graceful degradation path.

The resulting constraints are passed to **ConstraintChecker**, which multiplies each suggestion’s confidence by a penalty factor for each incompatible attack condition it detects. Attacks that are structurally impossible receive a penalty of 0.1, which reduces their score to a level at which they will not survive into the shortlist passed to the BDI loop. Attacks that are unlikely but not impossible receive intermediate penalties of 0.2 or 0.3. After the penalty pass the suggestions are re-sorted and the top five are retained, before the fuser assembles the final shortlist described in Section 4.2.5.

4.2.5 Information Fusion

The **InformationFuser** combines the filtered expert suggestions and the RAG output into a single ranked list of **Hypothesis** objects. The scoring formula for an expert suggestion is:

$$\text{score} = \text{confidence} \times q \times (1 - \alpha + \alpha \times h), \quad \alpha = 0.1$$

where *confidence* is the integer value (0–100) assigned by the CLIPS rule, q is a quality multiplier computed by the **ApproachEvaluator**, and h is a Laplace-smoothed historical success rate from the model’s persistent fusion history. The RAG hypothesis uses the same formula with the confidence term replaced by a fixed value of 80, which compensates for the absence of an explicit confidence signal from the retrieval server.

The quality multiplier q is computed without any LLM call. It is set to 0.0 if the approach text describes a brute-force search over more than one million operations, making it infeasible within the sandbox timeout; it is multiplied by 0.4 if the text contains template placeholders such as `[type]` or `<technique name>`, indicating that the RAG server returned a generic response; and it is multiplied by 0.3 if the response text is shorter than 80 characters. In all other cases $q = 1.0$.

The historical weight h is computed as:

$$h = \frac{w + K}{n + 2K}$$

where w is the number of past successful attempts for this technique category and n is the total number of past attempts, with the Laplace additive smoothing constant

$K = 1$. A technique with no history starts at $h = 0.5$, giving it a neutral weighting. One with a perfect record of 16 successes in 16 attempts reaches $h = 17/18 \approx 0.944$. The damping factor $\alpha = 0.1$ keeps the history term in the range $[0.9, 1.0]$, ensuring that historical evidence can only shift a score by at most 10%. This conservative choice is deliberate: the weight given to history is kept proportionate to the amount of evidence accumulated so far, so that a technique that happened to succeed or fail on the first few challenges does not dominate the ranking in a way that does not generalise.

To avoid coarse aggregation across dissimilar challenges, the history is tracked separately for each attack technique and, within a technique’s record, is filtered by **Jaccard tag similarity** before computing h . Only past attempts whose observation tags overlap with the current challenge’s tags by at least 0.3 are counted, and when no past attempt is similar enough the plain per-technique rate is used instead. This keeps the record a technique earned on one family of challenges from inflating its weight on a structurally different challenge that only shares a broad category tag.

The fuser applies one further rule to maintain diversity: the RAG hypothesis is guaranteed at least one slot in the final list regardless of its score, since a precise write-up match could outperform any general rule. The remaining slots are filled by the highest-scoring expert suggestions, with a maximum of two expert hypotheses to prevent a cluster of similar RSA rules from occupying the whole list at the expense of the write-up channel.

4.3 The Coordinator Agent

The **FirstAgent** class is the orchestrator. It holds references to all knowledge components and implements the end-to-end solving flow for a single challenge.

For each challenge the Coordinator agent first runs a **preliminary LLM extraction**, obtaining a structured JSON list of the vulnerabilities and security measures in the challenge text; this list is passed to the observation layer, the constraint analyser, and the RAG client, which use it to refine their outputs. It then attempts a **fast path**: the Coordinator agent calls the LLM with the full challenge text and asks it to produce the flag directly. This attempt is accepted only under a strict condition: if the challenge has a known ground truth in the system’s flag file, the fast-path response must match it exactly; if no ground truth is available, the fast-path response is always rejected. The reason for this asymmetry is that without a reference value a hallucinated answer is indistinguishable from a correct one, whereas an answer that matches the ground truth has been independently verified. Rejecting unverifiable fast-path answers forces the system to rely on computed evidence, that is, the output of code actually executed in the sandbox, rather than

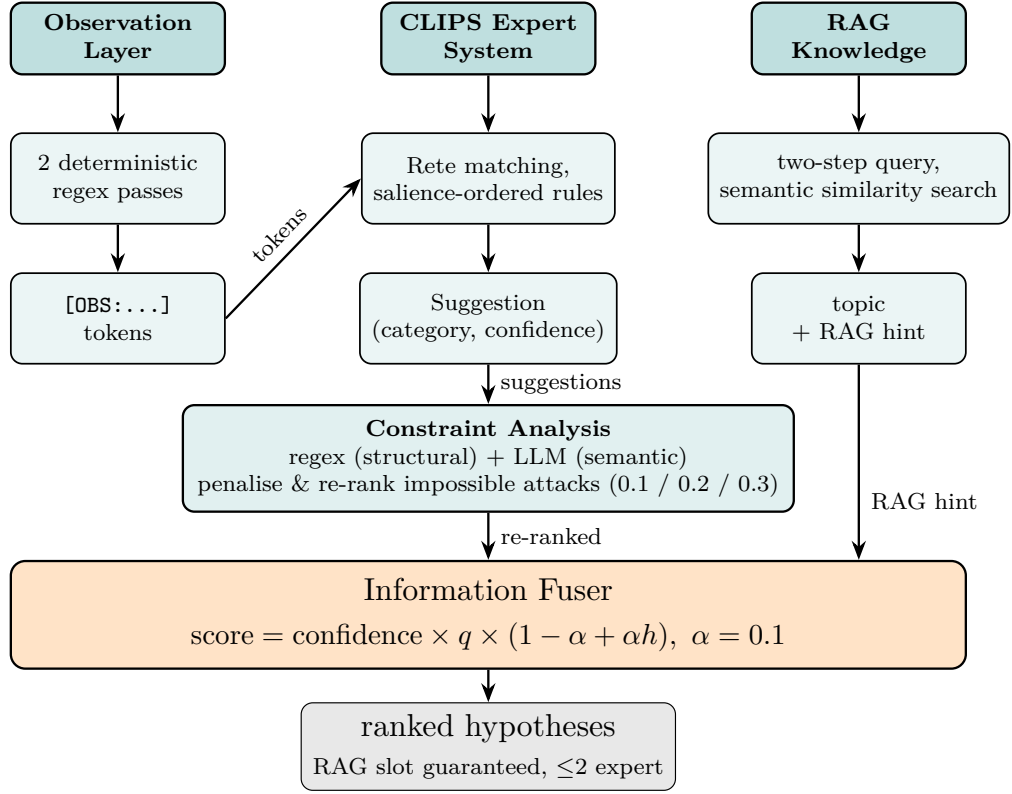


Figure 4.3: Internal detail of the knowledge-gathering stage, the constraint re-ranking, and the fusion step. The observation layer feeds structured tokens into the CLIPS expert system; both run without LLM calls, while the RAG server retrieves write-ups via one or more model calls. The constraint analyser (regex plus two LLM calls) does not feed the fuser directly: it penalises and re-ranks the expert suggestions that are structurally impossible before fusion. The Information Fuser then merges the RAG hint and the re-ranked expert suggestions into a single ranked hypothesis list, using the scoring formula of Section 4.2.5.

on the model’s assertions.

When the fast path is rejected, the Coordinator agent calls the function named `_gather_intelligence`, which runs the observation layer, the RAG query, the CLIPS engine, and the constraint filter in sequence, returning their outputs. The Coordinator agent then passes these outputs to the information fuser, which ranks them into the list of hypotheses.

The Coordinator agent then enters the **hypothesis loop**. For each ranked hypothesis it creates a fresh `TrueBDIAgent` instance and calls its `run` method. A fresh instance is used for every attempt so that beliefs accumulated during one approach cannot influence the next: beliefs are private to each attempt. The Coordinator agent waits for the BDI agent to terminate, then calls `_analyze_failure` to classify the outcome.

Failure classification is a two-stage process. The first stage applies a set of pre-compiled deterministic regex patterns to the execution output. Patterns for Python implementation errors such as `ImportError`, `SyntaxError`, `IndentationError`, `AttributeError`, and `TypeError` are matched against the output; each pattern has an associated hint builder that extracts the specific identifier from the exception message and returns a concrete correction instruction. Patterns for mathematical failures such as "no factors found", "wiener.*none", and "discrete log.*failed" trigger a verdict that the technique itself is wrong. When no regex pattern matches, the second stage calls the LLM with a prompt that includes the challenge text, the hypothesis label, the generated code, and the execution output, and asks for a JSON verdict. The LLM fallback is used only for ambiguous cases and is the minority path.

The verdict determines what happens next. If the failure is classified as an implementation bug (`keep_approach=True`), the same hypothesis is retried with the hint appended to the BDI requirements; this can happen at most `_MAX_RETRIES_PER_HYP = 2` times per hypothesis so that a plausible but ultimately wrong approach cannot exhaust the whole solving budget. If the failure is classified as a wrong technique (`keep_approach=False`), the outcome is recorded in the fusion history and the loop advances to the next hypothesis.

The Coordinator agent also exposes three environment-variable **ablation toggles**: `ENABLE_RAG`, `ENABLE_EXPERT`, and `ENABLE_CONSTRAINTS`. Setting any of these to `false` disables the corresponding knowledge source, allowing controlled experiments that measure the marginal contribution of each component to the overall solve rate.

The **anti-hallucination filter** applied to the output of each successful execution deserves separate attention. When the BDI agent reports that the code produced an output, the Coordinator agent does not accept it blindly as the flag. The filter applies three rejections in sequence. First, any output containing template placeholders such as `{flag}`, `<flag>`, or the literal string `FLAG` is rejected as a probable case in which the model printed the format string rather than the value. Second, any output that is a raw non-negative integer without any alphanumeric content beyond digits is rejected, since a real CTF flag is a meaningful byte sequence and a bare integer is almost certainly a partially computed intermediate value such as the plaintext m before byte decoding. Third, when a known flag for the challenge is stored in the system's flag file (`data/flags.txt`), the output must match it exactly; this is the check that actually confirms a solve, catching cases where the code ran without error but computed the wrong value, as happens with numerical hallucinations. An output that passes all three filters is accepted as the flag and the challenge is marked solved.

4.4 The BDI Agent

The `TrueBDIAgent` implements the BDI architecture described in Section 2.6, extended with the concrete beliefs, desires, and plans required for the code-generation task. The agent runs a loop of up to 30 cognitive cycles over a single hypothesis, and the pipeline from code generation to correct execution emerges from the interaction of beliefs, desires, and plans rather than from a fixed call chain [14].

Cognitive infrastructure

The **belief base** is a key-value store maintained by the `BeliefBase` class. Each entry maps a string key to a value of any type, and conditions are expressed as strings of the form `key=true`, `key=false`, `key!=None`, or bare `key` (existence check). The belief base records the full state of the current attempt: whether requirements have been understood, whether code has been generated and validated, whether the last execution succeeded, whether errors are present and debugging is needed, and the actual values of the generated code and the last execution output.

The **desire base** holds seven registered desires, each defined by three sets of conditions. The trigger conditions specify when the desire is active (what must be true for it to be a candidate). The preconditions specify what must be in place for the corresponding plan to start. The satisfaction conditions specify when the desire is already met and should be skipped. The seven desires, in order of decreasing priority, are:

- `fix_syntax_errors` (CRITICAL): active when the AST parse has failed. Fires before any other desire, including the debug loop.
- `debug_code` (CRITICAL): active when execution has produced errors and the code has no syntax problems. Repairs the code by calling the LLM with the error output and a concrete fix hint.
- `understand_requirements` (HIGH): active when user input has been received but not yet processed. Reads the hypothesis description and populates the belief base with the challenge parameters.
- `generate_code` (HIGH): active when requirements have been understood and no code exists yet.
- `execute_code` (HIGH): active when code has been saved and validated and there are no errors.
- `validate_code` (MEDIUM): active whenever code exists in the belief base. Checks it before execution.

- **save_code** (MEDIUM): active once code exists and saving has been requested. Writes the generated script to the workspace directory ahead of execution.

The CRITICAL priority level of the two repair desires ensures that the BDI agent never attempts to execute code that is known to be broken. When execution produces an error, **debug_code** fires before **execute_code** in every subsequent cycle until the code is repaired or the attempt is abandoned.

The **plan library** maps each desire to one or more plans, each of which is a sequence of named steps. The **intention structure** tracks which plans are currently active and at which step, and moves failed intentions to a separate list for diagnostics.

Code generation

The plan for **generate_code** consists of nine steps, organised as a chain of paired prompt-construction and LLM-call steps.

Steps 1–2 implement a **chain-of-thought pass**: the agent first asks the LLM to produce a numbered algorithm plan for the required cryptographic technique before writing any code. This commitment to a concrete mathematical plan before code generation is intended to reduce the rate at which the model writes code that is syntactically correct but algorithmically wrong. The plan is prompted at temperature 0 so that the algorithm description is conservative rather than creative. The rationale is that the step removes the decision burden from the code-writing phase: when the model generates code, it is following a concrete numbered plan rather than reconstructing the algorithm from the technique name.

Steps 3–4 generate an initial **code draft** conditioned on the algorithm plan. The prompt includes the full challenge text, the plan output, the list of libraries available in the sandbox, and a set of coding rules: the output must be raw executable Python that follows the plan exactly, every large integer must be hardcoded on its own line, and the solver must print only the bare recovered value with no flag prefix, since the Coordinator agent attaches the prefix deterministically after extraction. When the challenge involves a server, a further rule forbids network calls such as `pwntools.remote()` or `socket.connect()`. The library list is enumerated explicitly, spanning `pycryptodome`, `gmpy2`, and `sympy` for RSA up to `fp111` and `py_ecc` for lattice and elliptic-curve work, because smaller models are prone to importing standard modules that use fixed-width integers rather than arbitrary-precision equivalents, leading to silent integer overflow on large RSA moduli.

Steps 5–7 address a systematic weakness of the 7B model: Qwen 2.5 at 4-bit quantisation produces code with inconsistent indentation more frequently than larger models. The agent applies a **reformatting pass** that asks the LLM to

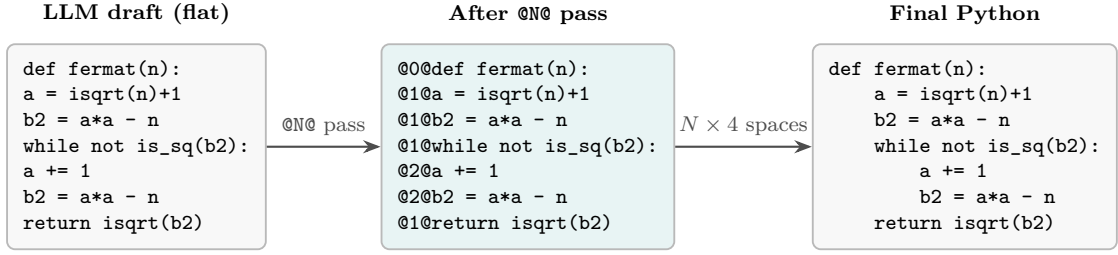


Figure 4.4: The @N@ reformatting pass on one function fragment. The LLM draft (left) has collapsed indentation; the reformatting call rewrites it with an explicit integer depth marker on each line (centre); `_replace_indent_markers` then turns each @N@ prefix into $N \times 4$ leading spaces, yielding valid Python (right).

rewrite the draft using an @N@ marker system where each line is prefixed with the integer depth of its nesting level. For example, a statement inside a function body at indentation depth 1 is prefixed @1@, and a statement inside a nested loop at depth 2 is prefixed @2@. The parser then converts each @N@content line into $N \times 4$ spaces followed by content. Explicit numeric markers remove the model’s need to produce the correct number of leading space characters, which is the step most prone to error under quantisation. A fast-path skip is applied when the draft already parses as valid Python, avoiding the extra LLM call entirely when the first draft is already correctly indented.

Figure 4.4 shows the same function fragment at each stage of this pass.

Step 8 applies targeted **domain micro-checks**: a set of pre-compiled regex patterns that detect specific common mistakes in cryptographic Python code. The patterns catch using `math.sqrt` on a large integer instead of `gmpy2.isqrt` (which overflows silently), writing `pow(n, d, n)` instead of `pow(ct, d, n)` for RSA decryption, wrapping a bytes index in `ord()` or `.decode()` (a Python 3 type error, since `bytes[i]` is already an integer), and treating `os.urandom` output as the known-plaintext prefix of an XOR attack. Each pattern triggers a focused single-prompt LLM correction call only when the corresponding bad pattern is present in the generated code, so the step adds zero LLM calls when the code is already clean.

Step 9 normalises the **output format**. A small classification call first identifies the type of the final computed value, an integer, bytes, a hex string, or a plain string, and a targeted rewrite then adjusts the closing `print` so that the solution is emitted as a decoded string. This prevents an output such as `b'CTF{...}'`, a `bytes` object printed without decoding, which would fail the exact flag match.

Validation and execution

The `validate_code` desire runs three sub-steps without any LLM call. First, an AST parse (`ast.parse`) checks whether the code is syntactically valid Python. Second, a host-side `pylint` invocation checks for undefined names, bad imports, and structural problems such as calls to `input()` or `sys.stdin`. Third, a structural pass rejects code that mixes tabs and spaces, since this produces indentation errors in the CPython interpreter that are otherwise hard to diagnose from the error message alone.

When `validate_code` completes without errors, the `save_code` desire fires and writes the script to the `workspace/` directory; once the code is both saved and validated, the `execute_code` desire fires in the next cycle. Execution is delegated entirely to the `SystemOperatorAgent`, which copies the file into the Docker container and runs it; the BDI agent perceives only the output that the `Explorer` agent returns.

Debugging

When execution fails, the `debug_code` desire activates. Its plan extracts the error output from the belief base, checks for specific patterns that require special treatment (a `ModuleNotFoundError` triggers a hint about which libraries are pre-installed in the sandbox; raw bytes or integer output triggers a conversion instruction), and calls the LLM at temperature 0.5 to explore a wider range of fixes. After the LLM produces corrected code, the plan resets the `code_validated`, `code_saved`, and `code_executed` beliefs to false so that the validate-save-execute pipeline re-runs on the new code.

Termination

An attempt ends when any of five conditions holds: the code has executed successfully; the code has not parsed correctly after 14 cycles; no desire remains active and no intention is pending; the cycle budget of 30 is exhausted; or 15 minutes of wall-clock time have elapsed. An additional fast-abort applies when the solver times out inside the container (exit code 124): in this case the agent records a timeout verdict immediately, skips the debug loop, and returns to the `Coordinator` agent, which advances to the next hypothesis rather than wasting another timeout window on the same slow code.

Algorithm 1 summarises the perceive-deliberate-act loop at the level of the desire selection mechanism, and Figure 4.5 shows the same cycle visually.

Algorithm 1 BDI perceive-deliberate-act cycle for one attempt

Require: hypothesis h , belief base B , desire set D , plan library P

Ensure: execution outcome $o \in \{\text{SUCCESS}, \text{FAILURE}, \text{TIMEOUT}, \text{BUDGET}\}$

```

1: initialise  $B$  with requirements extracted from  $h$ 
2: cycle  $\leftarrow 0$ 
3: while cycle < MAX_CYCLES and not terminated( $B$ ) do
4:   perceive environment  $\rightarrow$  update  $B$ 
5:    $A \leftarrow \{d \in D \mid \text{trigger}(d) \text{ satisfied in } B\}$ 
6:    $A \leftarrow \{d \in A \mid \text{preconditions}(d) \text{ satisfied in } B\}$ 
7:    $A \leftarrow \{d \in A \mid \neg \text{satisfied}(d, B)\}$ 
8:   if  $A = \emptyset$  then break
9:   end if
10:  intention  $\leftarrow \arg \max_{d \in A} \text{priority}(d)$ 
11:  plan  $\leftarrow \text{SELECT\_PLAN}(\text{intention}, P, B)$ 
12:  execute next step of plan  $\rightarrow$  update  $B$ 
13:  cycle  $\leftarrow \text{cycle} + 1$ 
14: end while
15: return outcome recorded in  $B$ 

```

4.5 End-to-End Pipeline Trace

To make the interaction between components concrete, this section traces the system’s handling of a representative RSA challenge. The challenge provides a 2048-bit modulus n , a public exponent $e = 65537$, and a ciphertext c , with the encryption implemented using a script that generates p and q by finding consecutive primes from a random starting point, so that p and q are close to each other.

Step 1: preliminary extraction. The Coordinator agent first calls the preliminary LLM extraction, which returns the JSON list ["RSA", "close primes", "Fermat factorization"].

Step 2: fast path. The Coordinator agent then submits the full challenge text to the LLM with a direct flag request. The model returns a plausible-looking flag string, but it does not match the actual flag stored in the challenge’s ground-truth file, so the fast path is rejected.

Step 3: intelligence gathering. With the fast path rejected, the Coordinator agent runs `_gather_intelligence`. The observation layer passes the challenge text through its regex patterns, identifies the substring `nextprime` in the source

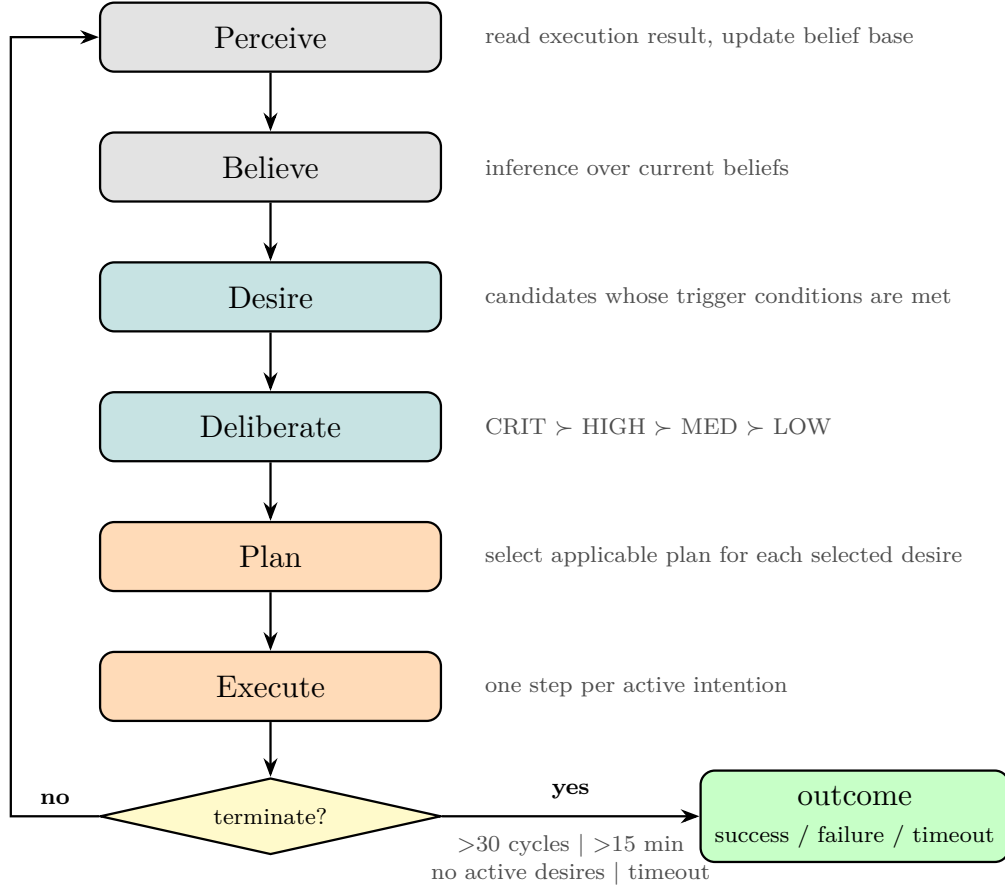


Figure 4.5: The BDI cognitive cycle as implemented in `TrueBDIAgent`. Each loop iteration advances every active intention by one plan step. Repair desires (`fix_syntax_errors`, `debug_code`) at CRITICAL priority pre-empt all others, so a cycle in which the previous execution produced errors is always a repair cycle.

code, and emits the `[OBS:CLOSE-PRIMES]` token, which is appended to the fact string handed to CLIPS.

CLIPS starts from the augmented fact and fires three RSA rules. The Fermat factorisation rule (salience 85, confidence 92) fires because the uppercased challenge source contains `nextprime`, which the observation layer also surfaces as the `[OBS:CLOSE-PRIMES]` token; the rule matches the literal `NEXTPRIME` marker directly. The `rsa-factordb` rule (salience 95, confidence 80) fires on the generic RSA markers (an exponent and a ciphertext), proposing a factordb lookup with a Pollard-rho fallback. The `rsa-rsactftool` rule (salience 60, confidence 52) fires last as a catch-all fallback. The small- n rule does not fire, since no small-modulus token is present for this 2048-bit challenge.

The RAG server returns a structured response citing a write-up from a previous

challenge with a near-identical generation pattern. The response quality passes the length and placeholder checks, so its score is 80 (confidence) $\times$ 1.0 (quality) $\times$ 0.95 (history term $1 - \alpha + \alpha h$ at neutral history) = 76.

The constraint analyser finds no constraints that would penalise the Fermat rule: the challenge provides one ciphertext and one modulus, and the code contains `nextprime`, not a multiple-recipient broadcast pattern.

The fuser scores the expert suggestions with the formula of Section 4.2.5: Fermat $\approx$ 92, `rsa-factordb` $\approx$ 76, `rsa-rsactftool` $\approx$ 48. The two-expert cap keeps only the two highest, Fermat and factordb, and drops the RsaCtfTool fallback; the RAG hypothesis ($\approx$ 76) takes the guaranteed third slot. The constraint filter penalises none of these. The final shortlist passed to the BDI loop is therefore [Fermat, factordb, RAG], with Fermat ranked first.

Step 4: BDI attempt on Fermat. The Coordinator agent creates a fresh `TrueBDIAgent` with the Fermat hypothesis and enters the cognitive loop, which advances the active intention by one plan step per cycle.

The agent first commits to `understand_requirements`, whose single step reads the hypothesis text and sets `requirements_understood=true`. It then commits to `generate_code`, whose nine-step plan runs over the following cycles: steps 1–2 produce a five-step Fermat algorithm plan (the chain-of-thought pass); steps 3–4 generate the code draft conditioned on that plan and the challenge values; steps 5–7 are the @N@ formatting pass, skipped here because the draft already parses as valid Python; step 8 finds no domain micro-bug to correct; and step 9 normalises the final `print`. The solver stored in the belief base is shown below:

`fermat_solver.py`

```

1  import gmpy2
2
3  def main():
4      n = 243823...    # 2048-bit modulus from challenge
5      e = 65537
6      c = 189017...    # ciphertext from challenge
7
8      a = gmpy2.isqrt(n) + 1
9      b2 = a * a - n
10     while not gmpy2.is_square(b2):
11         a += 1
12         b2 = a * a - n
13     b = gmpy2.isqrt(b2)
14     p = int(a - b)
15     q = int(a + b)
16     d = int(gmpy2.invert(e, (p - 1) * (q - 1)))
17     m = pow(c, d, n)

```

```
18     print(m.to_bytes((m.bit_length() + 7) // 8, 'big').decode())
19
20 main()
```

Once the code is generated, `validate_code` and `save_code` run at MEDIUM priority: the AST parse succeeds, pylint finds no undefined names, and the structural pass detects no mixed indentation, so `code_validated=true`; the script is written to the `workspace/` directory, so `code_saved=true`. Finally `execute_code` fires: the Explorer agent writes the Python file to the Docker container workspace, runs it, and returns its standard output. Because the step-9 normalisation rewrote the final `print`, this output is already a decoded ASCII string rather than the raw integer `m`.

The Coordinator agent runs the anti-hallucination filter on this output: it contains no template placeholder, it is not a bare integer, and it matches the stored ground-truth flag exactly. The flag is accepted, the challenge is marked solved, and the FusionHistory increments the win count for the Fermat technique.

4.6 The Explorer Agent

The `SystemOperatorAgent` is the embodied component of the system, responsible for all interaction with the execution environment. While the BDI agent reasons about what code to write, the Explorer agent reasons about how to deploy and run that code inside the sandbox.

Internal structure

The Explorer agent is organised as four internal layers arranged in a fixed invocation order (Figure 4.6). The **PerceptualCortex** scans the Docker container filesystem on each call: it lists the files in `/workspace`, records recently modified files, and maintains a rolling buffer of the last ten perception states to track changes across invocations. The **JSONLogicPlanner** receives the current perception state and the intent string from the BDI agent, sends them to an LLM, and parses the response as a JSON action specification. The planner applies three escalating fallback strategies when the LLM returns malformed JSON, and falls back to a regex parser for the most common patterns when all JSON recovery attempts fail. The **SafetyGuardrail** checks the planned action against a list of dangerous patterns before it is executed; it blocks any action that would delete system directories, write to protected paths, or invoke commands associated with destructive operations. The **MotorCortex** dispatches the validated action to the appropriate `DockerExecutor` method.

The action catalogue has nine entries: reading a file from the container, creating a file in the container, running a shell command, changing the working directory, taking a detailed observation of the environment, and searching the workspace, alongside three introspective actions that display the agent’s memory, report its status, and request clarification. The `DockerExecutor` implements each action by running `docker exec` against the named container. File transfer uses base64 encoding so that arbitrary binary content is safe to transmit. Every command execution wraps the inner command with `Linux timeout N` so that a slow script is killed inside the container before the host-side subprocess timeout fires; the inner timeout is 5 seconds shorter than the outer one to guarantee the container process exits cleanly. The command output is also capped at a fixed number of lines, so that a solver that spams its output in a tight loop cannot exhaust host memory before the process is killed.

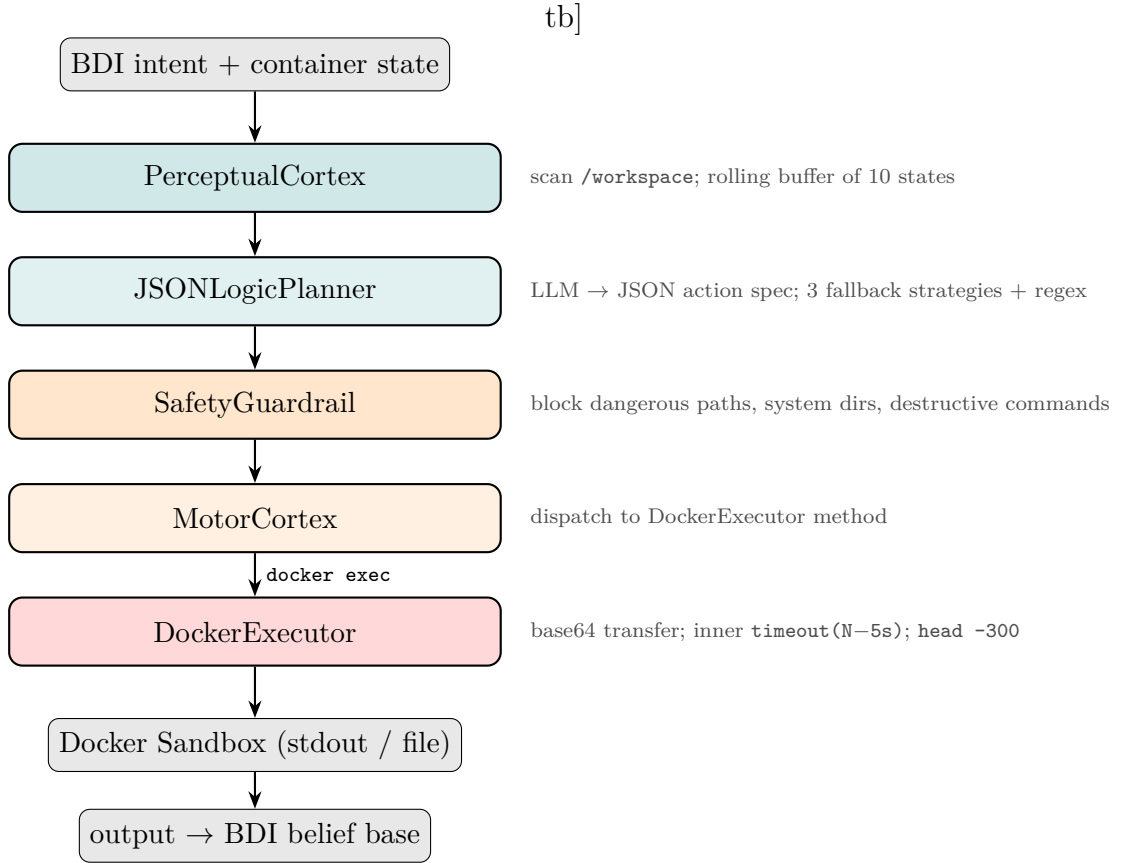


Figure 4.6: Internal pipeline of the Explorer Agent (`SystemOperatorAgent`). Each call from the BDI agent traverses all four layers in sequence. The `JSONLogicPlanner` is the only layer that calls an LLM; all others are deterministic.

Sandbox isolation

The safety guardrail described above is only a first line of defence against the accidental destructive commands that the model may generate, and it is not a security boundary in itself. Whatever it fails to intercept, including deliberate attempts to bypass it, is contained by the Docker sandbox, which is configured for defence-in-depth. The container filesystem is mounted read-only except for the `/workspace` volume, which is shared with the host, and a `tmpfs` mount at `/tmp` that provides a writable scratch area in RAM. All Linux capabilities are dropped. The `pids_limit` is set to 256 to prevent fork bombs. Memory is capped at 2 gigabytes and CPU at 1 core. The solver runs as a non-root user with a uid of 1000, set at image build time in `Dockerfile.sandbox`. The relevant portion of `docker-compose.yml` that enforces these constraints is:

```

docker-compose.yml
1  ctf-sandbox:
2    build: { dockerfile: docker/Dockerfile.sandbox }
3    volumes:
4      - ./workspace:/workspace          # shared with host (output)
5      - ./challenges_CTF:/challenges:ro
6    mem_limit: 2g
7    cpus: "1.0"
8    cap_drop: [ALL]                     # drop all Linux capabilities
9    security_opt: [no-new-privileges:true]
10   pids_limit: 256                      # no fork bombs
11   read_only: true                      # immutable root filesystem
12   tmpfs: [/tmp]                        # RAM scratch, wiped on stop
13   environment:
14     - HTTP_PROXY=http://egress-proxy:3128 # forced through Squid
15     - HTTPS_PROXY=http://egress-proxy:3128
16     - PYTHONDONTWRITEBYTECODE=1
17   networks: [sandbox_internal]         # internal only, no internet
18
19 networks:
20   sandbox_internal:
21     internal: true                     # no gateway to host or internet

```

The network configuration provides the strongest isolation boundary. The `ctf-sandbox` container is attached to an internal Docker network with no direct external routing. The only path to the internet goes through the **Squid egress proxy** (`egress-proxy`), which is the sole container with external connectivity. The proxy runs a deny-by-default policy: it consults an allowlist file and forwards requests only to two destinations: `factordb.com` (for RSA factorisation lookups)

and `pypi.org` together with `pythonhosted.org` (for package installation at runtime). Any other destination is silently blocked. This design confines the impact of a compromised or adversarial solver script to a small and auditable set of outbound connections.

4.7 Determinism and Local Execution

A recurring theme in this work is the distinction between the parts of the system that are deterministic and the parts that are not.

The **deterministic layer** includes the observation layer, the CLIPS forward-chaining engine, the constraint analysis regex pass, the information fusion scoring formula, the BDI cycle logic, the validation steps, and the first stage of the failure classifier. Given the same challenge text and the same fusion history, all of these components produce identical output on every run. This is the layer that carries the strategic decisions: which technique to try, in which order, and when to abandon an approach. One caveat applies: the observation tokens and the constraint analyser are seeded in part by the preliminary vulnerability extraction, which is itself an LLM call, so the strategic layer is deterministic given that extraction rather than in isolation. The observation regex pass provides a deterministic floor that holds even when the extraction varies, and at temperature 0 the extraction is stable across runs in practice.

The **non-deterministic layer** includes the LLM calls that generate, format, and repair code; the preliminary vulnerability extraction that seeds the observation tokens and the constraint analyser; the RAG retrieval, which depends on the embedding similarity computed at query time; and the LLM fallback in the failure classifier. Even with the temperature set to 0, small floating-point differences across hardware runs or framework versions can produce different token sequences. This layer is where the implementation is generated.

The architecture is designed so that the non-deterministic layer is as narrow as possible. The LLM is used for code generation, formatting, and repair, but not for deciding which attack to attempt, not for computing scores, and not for stopping conditions. A full run on the same challenge with the same fusion history will always start from the same ranked hypothesis list and will terminate under the same conditions; only the code it tries to execute may differ.

The **LLMPool** contributes to this design by eliminating a source of unnecessary overhead. Because three agents potentially use the same model, a naive implementation would load 3 separate copies of the 7B model into VRAM, exhausting the memory of a consumer GPU. The **LLMPool** maintains a class-level singleton cache keyed on the triple (`model`, `device`, `quantization`). When the three agents are configured to use the same model, the pool returns the same **LLMClient** instance

to all three, so only one copy of the model occupies GPU memory at any time. The practical consequence is that the system runs on a single consumer GPU with 12 gigabytes of VRAM, which is within the hardware budget of an individual researcher or practitioner rather than requiring a data-centre allocation.

Finally, the use of a fixed 4-bit NF4 quantised model [9] loaded from a local directory means that the behaviour of the model weights is stable across runs on the same hardware. There are no remote API endpoints whose model version can change overnight, no rate limits that alter which requests complete, and no terms-of-service changes that might restrict which challenges the system is allowed to attempt.

Chapter 5

Evaluation

This chapter presents the experimental evaluation of the proposed system. The goal is not to claim a state-of-the-art result but to answer the three research questions that guide the thesis (Section 1.5). First, to what extent a small local model, embedded in the architecture, can solve challenges that it cannot solve on its own. Second, how much of the resulting capability comes from the architecture rather than from the scale of the model. Third, where the residual failures concentrate, in the choice of the wrong cryptographic technique or in the incorrect implementation of a correct one.

The evaluation is structured as a comparison across three scenarios, from a raw baseline with no scaffolding, through an external commercial agent, to the proposed architecture, run on the same fixed dataset of challenges. Within Scenario 3, additional experiments vary the language model to separate the contribution of the architecture from the contribution of model scale, and a qualitative analysis examines the role played by each knowledge source.

5.1 Dataset

The dataset consists of 20 cryptographic CTF challenges. Each challenge is presented as a plain text file containing a problem description, source Python code that implements the encryption or key generation procedure, and the numerical parameters produced by running that code, ciphertexts, moduli, public keys, encrypted messages, and similar values. For each challenge the dataset also contains the correct flag, which is used as the verification oracle during evaluation.

While most of the challenges are drawn from past CTF competitions, a subset of 9 CTFs was created ad hoc with the support of AI, leveraging the author’s practical experience. These challenges cover the primary families of cryptographic weaknesses encountered in such competitions. Vulnerabilities in RSA implementations represent

the largest group (12 of the 20 challenges), followed by 5 XOR stream-cipher weaknesses, and 3 OpenSSL construction tasks that require a direct computation rather than an attack. Table 5.1 gives the full breakdown by technique.

No challenge requires network access or external services beyond those explicitly allowed by the sandbox policy. No challenge involves binary exploitation or reverse engineering: the evaluation is scoped exclusively to cryptographic challenges, because this is the category for which the symbolic knowledge layer encodes domain-specific rules. This scoping is a deliberate design choice, and its consequences for the system’s generality are discussed among the limitations in Chapter 6.

The table also records the attack/construction split introduced in Chapter 2. 17 of the 20 challenges are **attack** challenges, which expose a cryptographic weakness that must be recognised and exploited; the remaining 3 are **construction** challenges, for which every value needed is already present and the answer follows from a direct computation. The 3 construction tasks act as a deliberate control: the attack-hypothesis pipeline is not designed for them, so they probe the boundary of what the system is built to solve.

Family	Technique / weakness	Count	Type
RSA	Small modulus / FactorDB	3	Attack
RSA	Close-prime (Fermat)	2	Attack
RSA	Shared prime (GCD)	2	Attack
RSA	Small exponent $e = 3$	1	Attack
RSA	Hastad broadcast ($e = 3$)	1	Attack
RSA	Common modulus	1	Attack
RSA	Wiener (small d)	1	Attack
RSA	Pollard $p - 1$	1	Attack
XOR	Single-byte key	1	Attack
XOR	Rotating / repeating key	3	Attack
XOR	Two-time pad	1	Attack
OpenSSL	RSA parameter recovery	1	Construction
OpenSSL	ChaCha20 decryption	1	Construction
OpenSSL	Bitwise reconstruction	1	Construction
Total		20	17 / 3

Table 5.1: Challenge dataset: distribution across families and techniques, with the attack/construction split (17 attack, 3 construction).

5.2 Metrics

The primary metric is the **solve rate**: the fraction of challenges for which the system returns the correct flag within the allotted time. A flag is considered correct

only if it matches the stored ground truth exactly, after stripping whitespace. Partial credit is not awarded; a flag that is off by a single character counts as a failure. This binary criterion matches the evaluation semantics of real CTF competitions.

Secondary metrics are computed over the runs of Scenario 3 to characterise the quality of the system’s internal behaviour:

- **Per-technique success rate:** for each attack technique, whether proposed by the CLIPS expert system or retrieved through the RAG channel, the fraction of BDI attempts that produced the correct flag. This is tracked by the FusionHistory component and persists across runs.
- **Failure taxonomy:** each unsolved challenge is assigned to one of four categories:
 - *implementation failure:* the correct technique was selected but the generated code was defective, through a logic error, a wrong library call, or a mis-transcribed literal;
 - *construction outside the attack pipeline:* the challenge requires a direct computation rather than an attack, so the hypothesis pipeline never targets the real solution;
 - *topic misclassification:* the challenge was routed to the wrong rule set by the topic classifier;
 - *strategy budget exhaustion:* the correct technique was present in the ranked list but the attempt budget ran out before it was reached.
- **Hypothesis rank at success:** the rank of the hypothesis that produced the flag in the sorted list returned by the fuser. A low rank (first or second) indicates that the symbolic knowledge layer prioritised the correct technique; a high rank indicates that the correct technique was reached only after higher-ranked alternatives failed.

5.3 Experimental Setup

All experiments are run on a single machine with a consumer-grade NVIDIA RTX 3060 GPU (12 gigabytes of VRAM). Qwen 2.5-Coder-7B-Instruct [9] is loaded in 4-bit NF4 quantisation via the `bitsandbytes` library [10], occupying about 4 gigabytes for its weights, comfortably within the card’s 12-gigabyte budget. The CLIPS engine, the Gradio RAG server, and the Docker daemon run on the CPU and system memory and do not consume VRAM. The machine runs Linux with Docker Engine and a Squid proxy container.

All three agents are configured to use the same local model instance via the `LLMPool` singleton described in Section 4.7. Only one model copy is loaded. The maximum number of BDI cycles per hypothesis is 30, the per-attempt wall-clock timeout is 15 minutes, and the execution timeout for each generated script is 100 seconds.

The Docker sandbox image is built from a base Python 3.11 image with the standard cryptographic Python libraries pre-installed, covering RSA and AES primitives, arbitrary-precision integer arithmetic, symbolic computation, and HTTP queries to factorisation services. The image is fixed across all experiments; runtime package installation via `pip` is allowed only for the small set of domains in the Squid allowlist.

The RAG server runs locally as a Gradio application, reached by the Coordinator agent over HTTP at `127.0.0.1:7860`; its language model, however, is served through the OpenAI API rather than the local model, which makes it the system’s single non-local component. It maintains a vector index over a corpus of CTF write-ups and, given a challenge description, retrieves the most relevant ones through the two-step query protocol described in Section 4.2.3. The index is built once before the experiments begin and is not modified during the evaluation runs, so retrieval is stable across runs.

The CLIPS knowledge base `ctf_kb3.clp` is loaded fresh for each challenge. The Python wrapper resets the CLIPS environment between challenges to ensure that facts from one challenge do not leak into the next. The wrapper runs synchronously within the Coordinator agent’s process with no inter-process communication overhead.

For Scenario 3 with GPT-4o-mini, the entire language-model backbone is replaced: the Coordinator agent, the BDI agent, and the Explorer agent all run on GPT-4o-mini instead of the local Qwen model. The deterministic symbolic components, the CLIPS expert system, the constraint checker, and the information fuser, are model-independent and therefore identical across the two configurations, as is the RAG index. This configuration allows a comparison of the same architecture driven by a stronger model, with the symbolic layer held constant.

5.4 Scenario 1: Raw API Baseline

The first scenario establishes the lower bound. The challenge text is submitted to a language model in a single prompt with no scaffolding, no execution environment, no tool access, and no knowledge injection. The model is asked to return the flag directly. No code is generated; no computation is performed. The prompt includes only the challenge text and an instruction to output the flag on a single line or the string `NO_FLAG` if the answer cannot be determined.

Both models tested, Qwen 2.5-Coder-7B-Instruct and GPT-4o-mini, return 0 correct flags on the 20 challenges. The failure, however, is not one of understanding: on most of the 20 challenges both models identify a correct or partially correct approach, GPT-4o-mini in 18 cases (90%), 13 fully correct and 5 partial, and Qwen in 17 (85%), 12 fully correct and 5 partial. The entire gap is in turning that approach into a computed result. This result is unsurprising but informative. Cryptographic challenges require solving mathematical problems, factoring large integers, computing discrete logarithms, or recovering XOR keys, which cannot be solved by reading the problem statement and retrieving an answer from memory. The model can recognise the category of attack and describe the correct algorithm in natural language, but it cannot perform the underlying computation in its forward pass. Where the computation is trivial enough to attempt, such as recovering the XOR key from a two-character keystream, the model hallucinates a plausible-looking answer rather than computing the correct one.

The raw baseline therefore establishes that zero-shot prompting, regardless of model capability, is insufficient for this class of problems. Any positive result in subsequent scenarios can be attributed entirely to the scaffolding.

5.5 Scenario 2: External Commercial Agent

The second scenario provides a comparison with an external commercial agentic system. The same 20 challenges are submitted to OpenCode, a commercially available AI coding agent with tool-calling capability and an integrated code interpreter. This scenario is not a comparison of model architectures but of agent designs: both OpenCode and the proposed system rely on an LLM to generate code, but they differ in the knowledge layer surrounding the model and in the structure of the generation-validation-repair loop.

When given GPT-4o-mini as the backing model, OpenCode solves 2 of the 20 challenges: RSA Common Factor, the single easiest attack, recovered with one $\gcd(n_1, n_2)$, and Rand Byte-wise, a construction task that reduces to a direct bitwise computation. Both are the kind of short, well-known pattern most likely to appear in the model’s training data. OpenCode receives no structured hint about which technique to apply, relies entirely on the model’s parametric knowledge to select an approach, and has a generic debugging loop that is not specialised for cryptographic code. On the harder attacks it defaults to naive trial division, which is intractable for the moduli in the dataset, and on 3 XOR challenges it wrote correct code that the framework then failed to execute, logging them as unsolved.

When given Qwen 2.5-Coder-7B-Instruct, OpenCode solves 0 challenges. The 7B model does not reliably follow the tool-calling protocol that OpenCode requires: it conflates the JSON action format with natural-language output and produces

malformed tool calls that the orchestrator cannot parse. The agent enters a recovery loop but eventually exhausts its retry budget without producing any executable code. This result illustrates a practical limitation of generic agentic frameworks when applied to a model smaller than the frontier systems they were designed for.

The commercial agent therefore achieves a solve rate of 10% with its stronger model and 0% with the local model. This ceiling sets the reference point for the proposed architecture.

5.6 Scenario 3: The Proposed System

The third scenario runs the full proposed architecture: CLIPS expert system, RAG retrieval, structural constraint filtering, adaptive hypothesis ranking, BDI code-generation loop, and Docker sandbox execution, as described in Chapter 4.

Qwen 2.5-Coder-7B-Instruct. The system with the local model solves 11 of the 20 challenges, for a solve rate of **55%**. The raw single-prompt baseline and the commercial agent both solved zero on this same local model, so the entire gain is attributable to the architecture; against the commercial agent run on the stronger GPT-4o-mini, which solved 2, the local system solves more than five times as many.

The 11 solved challenges cover the techniques the architecture is strongest on: both Fermat close-prime challenges, the 3 small-modulus and FactorDB factorisations, the small-exponent cube root, the common-factor and common-modulus challenges, 2 of the 5 XOR challenges, and 1 of the 3 construction tasks. The 9 unsolved challenges break down as follows. Six are implementation failures, where CLIPS selected the correct technique but the generated code was defective: RSA Level 3, the Hastad broadcast, Pollard $p - 1$, the two-time pad, and the 2 repeating-key XOR challenges. Two are the construction tasks that the dataset includes as a deliberate control (Section 5.1): the answer is already present in the file and no attack applies, so failing them is the expected outcome and simply marks the boundary of what the attack pipeline is built to target. The last, the Wiener challenge, is a topic misclassification: it was routed to the PRNG rules, and because the sandbox error feedback never advertises the installed `owiener` package, the correct attack is never reached.

GPT-4o-mini. Running the whole pipeline on GPT-4o-mini instead of the local Qwen model raises the solve rate to 14 of 20, or **70%**, while the deterministic symbolic layer, the CLIPS expert system, the constraint checker, and the information fuser, stays identical. The change is concentrated in the implementation failures: GPT-4o-mini solves 4 challenges that Qwen had identified correctly but could not implement (RSA Level 3, the Hastad broadcast, Pollard $p - 1$, and the

two-time pad), in each case because it transcribes the large literals faithfully and produces more varied repair strategies across the debug cycles. The improvement is partly offset by one regression, RSA Common Modulus, which Qwen had recovered through a RAG hypothesis at the third rank but on which GPT-4o-mini exhausted its budget still trying to factor the modulus; the net effect is 3 additional solves. The 5 shared failures, namely the 2 construction tasks, the Wiener topic misclassification, and the 2 repeating-key XOR challenges, remain unsolved under both models, because they are pipeline-level issues rather than code-generation ones.

This gain confirms the intended decomposition rather than contradicting it. Although GPT-4o-mini now runs every LLM step in the pipeline, the deterministic strategic layer, which selects the technique and orders the attempts, is unchanged: the CLIPS rules, the constraint checker, and the fusion scoring produce the same ranking regardless of the model, and the topic-identification rate does not improve with GPT-4o-mini, being in fact marginally lower. The gain concentrates instead at the one step the architecture leaves open-ended, the writing of correct code: every additional solve is a challenge whose technique was already selected correctly but whose code Qwen could not get right. GPT-4o-mini does not therefore outperform the scaffolding; it runs inside it and improves that non-deterministic step, exactly as the failure taxonomy predicts. The 15-point gap thus measures the residual value of code-generation quality, and it is paid for with per-query API fees and with the transmission of challenge data to a third party, which the fully local configuration, at 55%, avoids.

5.7 Results and Discussion

Table 5.2 summarises the solve rates across all conditions.

Scenario	Model	Solve rate
Raw API baseline	GPT-4o-mini	0/20 (0%)
Raw API baseline	Qwen 7B 4bit	0/20 (0%)
OpenCode	GPT-4o-mini	2/20 (10%)
OpenCode	Qwen 7B 4bit	0/20 (0%)
BDI architecture	Qwen 7B 4bit	11/20 (55%)
BDI architecture	GPT-4o-mini	14/20 (70%)

Table 5.2: Solve rates across all experimental conditions on the 20-challenge dataset.

The results confirm the central claim of the thesis, that the architecture contributes more than the model, through the comparison between raw Qwen 7B (0/20), OpenCode with GPT-4o-mini (2/20), and BDI with Qwen 7B (11/20).

The same local model improves from 0% to 55% when placed inside the proposed architecture, while the better model in a generic framework reaches only 10%. The architecture, not the model scale, is the decisive factor.

A further finding, that the architectural gains are not due simply to having any agent, is confirmed by the OpenCode comparison. OpenCode is a capable commercial agent; it has tool access and a code interpreter. It still falls far short of the BDI system on the same model configuration. The difference lies in the structured knowledge layer: CLIPS rules provide high-precision technique selection, constraint filtering removes non-applicable hypotheses before any code is generated, and the BDI desire priorities direct the repair cycle to the most critical issue first rather than re-attempting from scratch.

A final finding, that residual failures are diagnosable and split between technique-selection errors and implementation errors, is confirmed by the failure taxonomy detailed in Table 5.3. For the local model the failures concentrate in implementation rather than technique selection: the symbolic layer selects the correct technique for 17 of the 20 challenges, missing only the 2 out-of-scope construction tasks and the 1 misclassified challenge, so the residual bottleneck is faithful code generation. The implication is that improving the code-generation model and broadening the pipeline’s scope address distinct subsets of the failures independently.

Failure taxonomy breakdown

Table 5.3 distributes the 9 Qwen and 6 GPT-4o-mini unsolved challenges across the failure categories defined in Section 5.2.

Failure category	Qwen 7B	GPT-4o-mini
Implementation bug (defective code)	6	2
Construction outside attack pipeline	2	2
Topic misclassification	1	1
Strategy budget exhausted	0	1
Total failures	9	6

Table 5.3: Failure distribution for the 9 (Qwen) and 6 (GPT-4o-mini) unsolved challenges in Scenario 3.

The table shows where a stronger code generator helps and where it does not. The upgrade from Qwen 7B to GPT-4o-mini cuts the implementation failures from 6 to 2: the 4 challenges whose technique Qwen selected correctly but could not implement are solved once the code is generated faithfully. The 2 repeating-key XOR challenges survive as implementation failures under both models, with GPT-4o-mini producing near-miss flags that are 1 or 2 bytes wrong rather than crashing.

The construction and topic-misclassification failures are unchanged, since they are properties of the pipeline rather than of the code generator. The one new failure under GPT-4o-mini, RSA Common Modulus, is a strategy-budget case: the correct RAG hypothesis was available, but the agent spent its budget on the wrong first hypothesis, whereas the Qwen run happened to reach the RAG hypothesis in time.

Relationship to the sovereign AI argument

The evaluation also bears on the operational-independence argument introduced in Section 2.3, and it is important to be precise about how far the system actually reaches that ideal. The reasoning core runs locally: with Qwen 7B, the vulnerability extraction, the CLIPS expert system, the constraint checking, the hypothesis ranking, the BDI cycle, and the code generation all execute on the local model and on the local hardware, at no per-query cost. One component is the exception. The retrieval module, reused from a previous thesis as a compiled executable, calls the OpenAI API internally and could not be reconfigured to a local model, so any run that consults it is not fully local. Because the retrieval module is part of the pipeline and is queried on every challenge, the reported figures are best read as two levels of independence: a **fully-local** configuration, using only the local model and the expert system, accounts for the 7 challenges solved through expert hypotheses, that is 35%; adding the external retrieval hints raises this to the reported 55%, since 4 of the 11 solved challenges are recovered through the retrieval channel. The retrieval module is therefore a minority contributor, and closing the remaining gap by replacing it with a local retriever is left as future work (Section 6.3). Even at 35%, the fully-local configuration far exceeds the raw baseline and the commercial agent on the same local model, so the architectural claim does not depend on the external component.

5.8 The Role of Each Knowledge Source

The system exposes three environment-variable flags, `ENABLE_RAG`, `ENABLE_EXPERT`, and `ENABLE_CONSTRAINTS`, that can disable each knowledge source independently, so a quantitative ablation is possible in principle. Such a solve-rate sweep was not run for this thesis; instead, this section reasons qualitatively about the role of each component from the per-challenge behaviour of the full-system runs, in particular which source contributed the winning hypothesis to each of the 11 Qwen solves. A full quantitative ablation is left as future work.

Disabling CLIPS (`ENABLE_EXPERT=false`). Without the expert system, the fuser receives only the RAG hypothesis and the LLM’s preliminary vulnerability

extraction. The preliminary extraction is a flat list of vulnerability names without confidence scores, action descriptions, or salience ordering. The system falls back to testing hypotheses in the order that the LLM returns them, which is not aligned with the expected solve probability for each technique. Challenges where the CLIPS rule assigned high confidence to the correct technique are the most affected: without the rule, the correct technique may rank lower than a competing hypothesis and the attempt budget may be exhausted before it is reached. The CLIPS component accounts for the correct first-ranked hypothesis in 7 of the 11 successful Qwen runs, with the remaining 4 recovered by RAG; removing the expert system would require the RAG or the preliminary extraction to cover those 7 cases, which they do not do consistently.

Disabling RAG (`ENABLE_RAG=false`). Removing the RAG component reduces the set of technique hypotheses to those generated by CLIPS alone. For challenges that have a close match in the write-up index, the RAG hypothesis sometimes carries a more detailed procedure description than the CLIPS action string, enabling more precise code generation on the first attempt. The effect is most visible on challenges where the correct technique is covered by a CLIPS rule but the rule’s action description is high-level, in which case the RAG write-up provides the additional implementation detail needed for the code to compile and run correctly on the first generation pass. Disabling RAG would therefore remove the 4 challenges that only the retrieval channel solved, and would cost extra generation attempts on the others where the write-up supplied implementation detail that the terse expert action string lacked.

Disabling constraint filtering (`ENABLE_CONSTRAINTS=false`). Without the constraint filter, hypotheses that are mathematically incompatible with the challenge parameters remain in the ranked list at their original confidence score. The practical consequence is that the BDI agent may attempt a technique that requires 3 ciphertexts on a challenge that provides only one, or a technique that requires a large number of PRNG outputs on a challenge that provides fewer. Each such attempt runs the BDI loop to exhaustion before the Coordinator agent advances to the next hypothesis. The constraint filter prevents these wasted attempts; removing it would let structurally impossible hypotheses consume attempts before the correct technique is reached, risking budget exhaustion on the challenges where the correct technique is not already ranked first.

5.9 Intermediate Experiments

Beyond the three main scenarios and this component analysis, the system’s internal FusionHistory component records the outcome of every BDI attempt and computes running per-technique success rates using the Laplace-smoothed estimator described in Section 4.2.5. These records accumulate across all runs on the dataset and provide a fine-grained view of where the system is reliable and where it is not.

Table 5.4 reports the per-technique success rates accumulated in the fusion history across the development runs with Qwen 2.5-Coder-7B-Instruct.

Family	Technique	Wins/Attempts	Rate
RSA	rsa-fermat (Expert)	16/16	100%
RSA	rsa-small-n (Expert)	23/26	88%
XOR	RAG (xor)	4/5	80%
XOR	xor-rotating-key (Expert)	7/9	78%
RSA	RAG (rsa)	8/14	57%
RSA	rsa-factordb (Expert)	7/13	54%
RSA	rsa-pollard-p1 (Expert)	1/5	20%
RSA	rsa-common-modulus (Expert)	0/4	0%
RSA	rsa-rsactftool (Expert)	0/2	0%

Table 5.4: Per-technique success rates accumulated in the fusion history across all development runs with Qwen 2.5-Coder-7B-Instruct. Totals exceed the 20-challenge count because the history persists across repeated runs.

Several observations follow from this table. The Fermat factorisation technique achieves a perfect record across 16 attempts. The corresponding code is structurally simple, a loop that increments a from $\lceil \sqrt{n} \rceil$ and checks whether $a^2 - n$ is a perfect square, which the 7B model generates correctly on the first attempt in every case. The `rsa-small-n` technique is also highly reliable at 88% (23 of 26); its 3 failures fall on challenges where the modulus is not actually small enough to factor directly, so the technique is mismatched to the challenge rather than mis-implemented.

The RAG-originated hypotheses achieve 57% on RSA challenges (8 of 14) and 80% on XOR challenges (4 of 5), lower than the strongest expert rules but well above chance. The write-up retrieval succeeds when the vector index contains a challenge sufficiently similar to the current one; when the index lacks a close match, the hypothesis quality degrades and the code generation fails. The `rsa-pollard-p1` technique is attempted 5 times but succeeds only once: the implementation requires the model to correctly write the prime-factorisation loop over a large bound, which is within the code complexity range where the 7B model frequently introduces off-by-one errors and incorrect modular arithmetic. The `rsa-common-modulus`

expert rule has a 0% success rate over 4 attempts, yet the challenge is still solved through the RAG channel: the expert action description was too terse for the model to implement, while the retrieved write-up carried the full Bezout procedure.

The `rsa-rsactftool` technique has a 0% success rate across both attempts. The CLIPS rule for this technique describes using the `RsaCtfTool` command-line utility, which is not installed in the Docker sandbox. Both attempts terminate with an error reporting that the tool is not found on the first execution, the failure classifier correctly labels them as `wrong_tool`, and the Coordinator agent advances to the next hypothesis. The 2 failures represent a knowledge-base inconsistency rather than a model limitation; the rule should either be removed from `ctf_kb3.clp` or the tool should be installed in the sandbox image.

The aggregate picture from the per-technique table is consistent with the failure taxonomy from the main experiment. Techniques with simple, formulaic implementations succeed reliably regardless of model size. Techniques with complex algebraic implementations fail more often with the smaller model. Techniques that depend on unavailable tools fail deterministically. These are three distinct failure modes that call for three distinct remedies: model upgrade, implementation simplification, and knowledge-base maintenance.

The per-technique data also reveals an asymmetry in how the FusionHistory learning signal propagates. Techniques with many attempts accumulate enough evidence for the Laplace-smoothed estimator to converge towards a stable weight. The Fermat and small- n techniques, with 16 and 26 attempts respectively, have weights close to their empirical rates. The Pollard- $p - 1$ technique, with only 5 attempts, has a smoothed weight of $(1 + 1)/(5 + 2) \approx 0.29$ rather than the naive $(1/5) = 0.20$; the smoothing conservatively pulls the estimate towards the neutral prior of 0.5. The consequence is that the Pollard rule is not over-penalised by its 5-attempt history, and will continue to be proposed when the CLIPS conditions match, giving the technique the opportunity to accumulate a more reliable estimate over additional runs. This behaviour is the intended effect of the Laplace additive prior: it prevents early failures from permanently suppressing a valid technique while still using accumulated evidence to adjust priorities for mature techniques.

Hypothesis rank at success

A secondary question about the system’s internal behaviour is how far down the ranked hypothesis list the Coordinator agent must reach before finding the correct technique. An ideal information fuser would always place the correct technique first; a random fuser would require, on average, a number of attempts equal to half the list length before reaching it.

Across the 11 successful runs with Qwen 2.5-Coder-7B-Instruct, the correct technique was ranked first in 8 cases, second in 1 case, and third in 2 cases. No

successful run needed to go beyond the third-ranked hypothesis, which is also the last slot in the fused shortlist, at most 2 expert hypotheses plus 1 RAG hypothesis. This distribution reflects the quality of the symbolic knowledge layer: CLIPS rules, informed by the observation tokens and filtered by the constraint checker, assign high confidence to the correct technique in most cases, placing it at the top of the fused list before any BDI attempt begins.

The 3 challenges not solved by the first-ranked hypothesis (XOR Single Byte at rank 2, Small-RSA and RSA Common Modulus at rank 3) were all recovered by RAG-originated hypotheses after the higher-ranked expert rules failed: the write-up retrieval surfaced a more precise procedure than the expert action string in each case. After these runs, the FusionHistory raised the success weight for the RAG-derived hypotheses, so subsequent challenges of the same type would see them ranked higher. This incremental adjustment illustrates the intended feedback loop between the hypothesis ranking and the accumulated solving history.

Chapter 6

Conclusion

This final chapter concludes the thesis by reflecting on the research objectives and the results obtained. It first summarises the primary contributions of the proposed neuro-symbolic architecture, highlighting how the structural design compensates for the limitations of a small local model. Subsequently, it acknowledges the current constraints of the system, outlines potential directions for future work, and provides closing remarks on the broader implications of operational independence in automated security tools.

6.1 Summary of Contributions

This thesis has presented a neuro-symbolic multi-agent architecture for the autonomous resolution of cryptographic CTF challenges, built around a small local open-weight model. The starting point was a specific observation, which goes beyond the mere truism that cryptography requires computation. Recognising which attack a challenge calls for and carrying it out are two different abilities. A model may name the right attack, and yet the decisive step is an exact computation that no single forward pass produces, so that the result must come from code that is actually executed. Powerful frontier models, given tools and enough attempts, now handle both abilities well; the harder and more interesting case is a small model, run locally, which identifies the attack just as readily but cannot, on its own, turn it into a correct, executed result. What such a model needs is not more scale, but a structure placed around it, capable of carrying a recognised attack through to an executed result. This is the reason why every system that reaches meaningful solve rates on this class of problems ends up wrapping the model in a loop, in which the code is generated, executed in a real environment, and then revised on the basis of the outcome.

The contribution of this work consists in showing that such a structure can

be built in a way that compensates for the smaller capacity of a local model, by rendering the strategic reasoning layer deterministic and explicit. In particular, an expert system and a retrieval module supply the cryptographic knowledge and rank the candidate attacks, so that the system already knows which technique to attempt before the model is invoked. The BDI agent, in turn, confines the non-deterministic model to the generation and the repair of the code, while it keeps every branching decision under deterministic control. The specific role of each component is described in detail in Chapter 4.

The empirical evaluation supports the central claim. When no scaffolding is provided, both the local model and the stronger commercial model solve none of the 20 challenges, while a commercial agent endowed with tool access reaches only 10% with the stronger model. The proposed architecture, by contrast, reaches 55% with the local 7B model, and 70% when the very same architecture is driven by the stronger model in place of the local one, the symbolic layer remaining identical in both cases. The fact that the local model rises from 0% to 55% inside the architecture, while a stronger model inside a generic agent reaches only 10%, constitutes the central evidence that the architecture contributes to the outcome more than the scale of the model. Moreover, the difference between the two models within the architecture is concentrated in the quality of the generated code.

The failure analysis, finally, locates the residual bottleneck. The selection of the technique is rarely the problem, since the symbolic layer identifies the correct attack for 17 of the 20 challenges. The failures that remain with the local model are dominated by defective generated code, rather than by a wrong choice of technique, and this is confirmed by the stronger model, which recovers most of them. Consequently, coverage and implementation turn out to be largely independent concerns, addressed respectively by the extension of the knowledge base and by the improvement of the code generation.

6.2 Limitations

Several limitations of the system should nonetheless be acknowledged.

In the first place, the **scope of the knowledge base** is confined to the cryptographic attacks that are encoded in the CLIPS rules. Other CTF categories, such as binary exploitation, web, or forensics, therefore fall outside it, and the evaluation dataset was chosen accordingly.

A second limitation concerns the **numerical hallucination** of the 7B model, which may mis-transcribe a multi-digit integer from the challenge into the generated code. This behaviour is the main cause of the remaining implementation failures, and it is only partly intercepted by the current checks.

More significant, with respect to operational independence, is the **external**

retrieval dependency. The reused retrieval module is in fact a black box served through the OpenAI API, so that the system remains local in its reasoning core but not in this single component. It is nevertheless a minority contributor, and the fully-local configuration still reaches 35%.

Finally, the **sandbox timeout** rules out techniques that are correct but slow. Certain parameterisations of Pollard’s $p - 1$ method, for instance, occasionally exceed the time limit even when the implementation itself is sound.

6.3 Future Work

The analysis points to several directions for improvement, most of which require no modification of the architecture itself:

- a **literal-extraction validator**, executed immediately after the generation of the code, which would compare every integer and string literal against the challenge text, thereby eliminating the numerical-hallucination failures that a larger model otherwise removes;
- the **replacement of the external retrieval module with a local one**, built around a local embedding model and a local language model, which would render the whole pipeline sovereign and would remove the only component that transmits challenge data outside the operator’s boundary;
- the **extension of the knowledge base**, in the near term with further cryptographic techniques such as lattice and elliptic-curve attacks, and then with rule sets for the other CTF categories. Since the reasoning core is not specific to cryptography, such an extension is largely a matter of adding knowledge rather than of redesigning the system;
- in the longer run, the extension of the system **from the offensive side alone to both attack and defence**, reasoning about how to harden a target as much as about how to break it, with the eventual aim of an autonomous, AI-driven penetration-testing tool that preserves the local, explicit, and sandboxed design of the present work.

6.4 Closing

The thesis opened with a question: can a local model, embedded in an appropriate architecture, solve security challenges that it cannot solve on its own? The answer, supported by the evaluation, is yes, and the margin is substantial. A 7B model that scores zero on direct prompting reaches 55% inside the proposed system. The

architecture that makes this possible is not exotic: it combines an established expert-system formalism, a retrieval component, a classical cognitive agent model, and a Docker sandbox, all available as open tools. The contribution is in how these components are composed and in the argument, both theoretical and empirical, that composing them in this way compensates for the gap in model scale.

The practical implication is that the choice between a local model and a frontier API is not simply a choice between capability and cost. It is a choice about what can be known, trusted, and controlled about the system. A system whose reasoning is grounded in explicit symbolic rules, whose execution is confined to an auditable sandbox, and whose model weights are fixed on local hardware is easier to inspect, correct, and trust than one that delegates its cognition wholesale to a remote service. The system described here realises this for its reasoning core; the single external component it still relies on is the natural next step towards full independence. For applications in which reliability, reproducibility, and operational independence matter, the architectural investment this thesis describes may be worth making.

Bibliography

- [1] Shalabh Kumar Singh and Shubhashis Sengupta. *Sovereign AI: Rethinking Autonomy in the Age of Global Interdependence*. arXiv preprint arXiv:2511.15734. 2025. arXiv: 2511.15734 [cs.CY] (cit. on pp. 2, 10, 21).
- [2] Anthropic. *Disrupting the First Reported AI-Orchestrated Cyber-Espionage Campaign*. Threat intelligence report. 2025. URL: <https://assets.anthropic.com/m/ec212e6566a0d47/original/Disrupting-the-first-reported-AI-orchestrated-cyber-espionage-campaign.pdf> (visited on 06/29/2026) (cit. on p. 2).
- [3] Talor Abramovich et al. «EnIGMA: Interactive Tools Substantially Assist LM Agents in Finding Security Vulnerabilities». In: *Proceedings of the 42nd International Conference on Machine Learning (ICML)*. Vol. 267. Proceedings of Machine Learning Research. 2025. arXiv: 2409.16165 [cs.AI] (cit. on pp. 3, 5, 18).
- [4] Minghao Shao, Boyuan Chen, Sofija Jancheska, Brendan Dolan-Gavitt, Siddharth Garg, Ramesh Karri, and Muhammad Shafique. *An Empirical Evaluation of LLMs for Solving Offensive Security Challenges*. arXiv preprint arXiv:2402.11814. 2024. arXiv: 2402.11814 [cs.CR] (cit. on pp. 3, 5, 10, 19).
- [5] Johan Håstad. «Solving Simultaneous Modular Equations of Low Degree». In: *SIAM Journal on Computing* 17.2 (1988), pp. 336–341. DOI: 10.1137/0217019 (cit. on p. 8).
- [6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017. arXiv: 1706.03762 [cs.CL] (cit. on p. 8).
- [7] Siboy Yi, Yule Liu, Zhen Sun, Tianshuo Cong, Xinlei He, Jiaying Song, Ke Xu, and Qi Li. «Jailbreak Attacks and Defenses Against Large Language Models: A Survey». In: (2024). arXiv: 2407.04295 [cs.CR] (cit. on p. 10).
- [8] Anthropic. *Redeploying Fable 5*. 2026. URL: <https://www.anthropic.com/news/redeploying-fable-5> (visited on 07/06/2026) (cit. on p. 10).

- [9] Binyuan Hui et al. *Qwen2.5-Coder Technical Report*. arXiv preprint arXiv:2409.12186. 2024. arXiv: 2409.12186 [cs.CL] (cit. on pp. 11, 45, 48).
- [10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. «QLoRA: Efficient Finetuning of Quantized LLMs». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36. 2023. arXiv: 2305.14314 [cs.LG] (cit. on pp. 11, 48).
- [11] Charles L. Forgy. «Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem». In: *Artificial Intelligence* 19.1 (1982), pp. 17–37. DOI: 10.1016/0004-3702(82)90020-0 (cit. on p. 12).
- [12] Henry A. Kautz. «The Third AI Summer: AAAI Robert S. Engelmore Memorial Lecture». In: *AI Magazine* 43.1 (2022), pp. 105–125. DOI: 10.1002/aaai.12036 (cit. on p. 13).
- [13] Michael E. Bratman. *Intention, Plans, and Practical Reason*. Cambridge, MA: Harvard University Press, 1987. URL: <https://philpapers.org/rec/BRAIPA> (cit. on p. 13).
- [14] Anand S. Rao and Michael P. Georgeff. «BDI Agents: From Theory to Practice». In: *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*. San Francisco, CA, 1995, pp. 312–319 (cit. on pp. 13, 34).
- [15] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33. 2020, pp. 9459–9474. arXiv: 2005.11401 [cs.CL] (cit. on p. 16).
- [16] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. «SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 37. 2024. arXiv: 2405.15793 [cs.SE] (cit. on p. 18).
- [17] Zimo Ji, Daoyuan Wu, Wenyuan Jiang, Pingchuan Ma, Zongjie Li, and Shuai Wang. «Measuring and Augmenting Large Language Models for Solving Capture-the-Flag Challenges». In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Taipei, Taiwan, 2025, 18 pages. DOI: 10.1145/3719027.3744855. arXiv: 2506.17644 [cs.AI] (cit. on p. 20).