Final Project for M.S. in Computer Science

Aryan Kumar

Advisor: Shambhu

Upadhyaya

University at Buffalo

August 10, 2026

TABLE OF CONTENTS

Final Project Report(Part 1)

Name: Aryan Kumar
Duration: Jan 25, 2026 – May 10, 2026
Supervisor Name: Shambhu Upadhyaya

---

1. Introduction

1.1 Background

AI's are now widely used in businesses to help employees complete tasks faster. Workers can ask AI systems to summarize reports, analyze data, or generate documents using normal language.

Although these tools improve productivity, they also create security risks. Companies need a way to monitor how AI is being used and whether employees are accessing or sharing sensitive information improperly.

Examples of possible risks include:

- Asking the AI to summarize confidential financial data
- Trying to export sensitive records
- Using AI to gather large amounts of internal information
- Prompt injection attacks(Greshake et al., [5])
- Data exfiltration through AI responses

Traditional security tools cannot fully understand these types of conversations because they only inspect files, keywords, or network traffic.

1.2 Project Motivation

The goal was to research and build a system that could:

- Analyze AI conversations in real time
- Understand user intent and risk
- Track behavior across multiple prompts
- Use identity and permission data when evaluating risk
- Create audit records for security teams
- Work without interrupting the user experience

2. Problem Statement

Enterprise AI systems create a governance gap. Existing tools can log conversations but cannot understand the meaning behind them.(OWASP Foundation, [4])

For example:

“Show vendor spending trends for Q1 2026.”

By itself, this may look harmless. However, if earlier prompts involved downloading large datasets or emailing reports externally, the same request could become suspicious.

Prompt Governance Intent Handler(PGIH) was designed to solve these issues using AI-powered governance analysis.

---

3. System Architecture

The PGIH architecture :

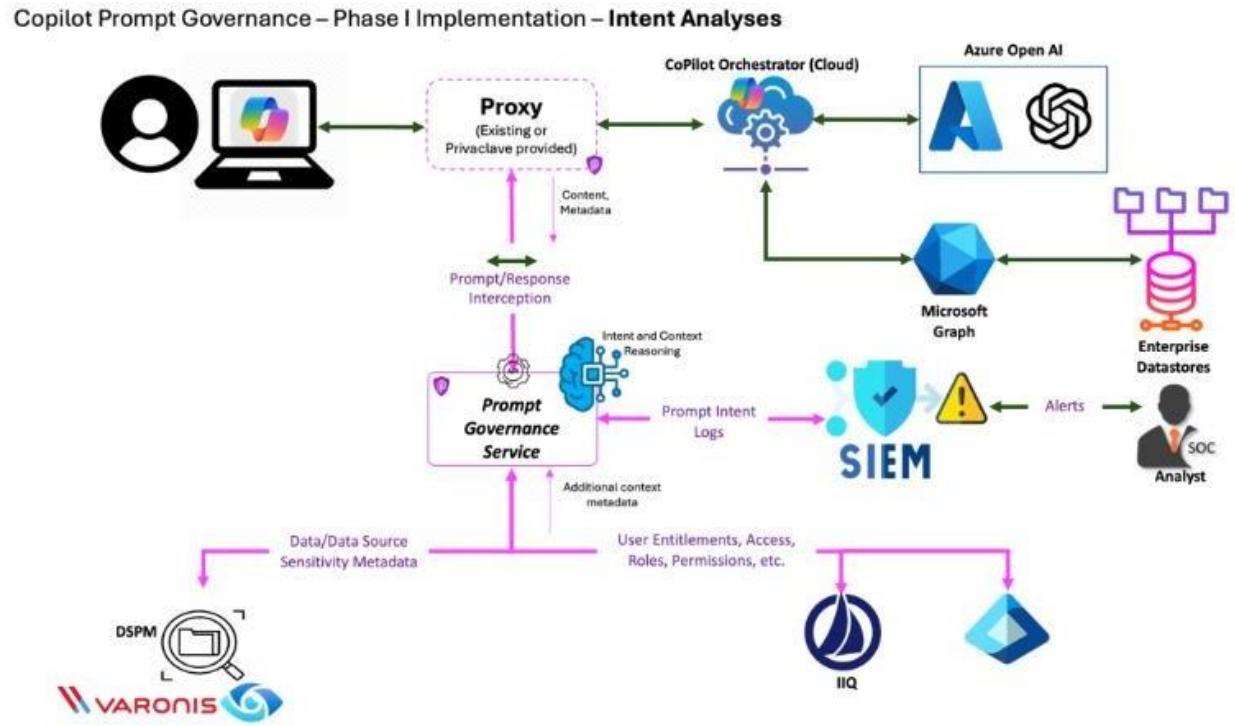

Fig 1: Architecture of the workflow

PIHS (Prompt Intent Handling Service)

Handles incoming prompts, stores events, and manages session flow.

PGCP (Policy and Governance Context Provider)

Resolves user identity, roles, and permissions from:

- Microsoft Entra ID
- SailPoint
- Varonis

AI Engine

Runs governance analysis using multiple small language models in parallel.

Turn Judge

Combines the outputs from the models into one final governance result for each turn.

Session Judge

Creates a session-level summary after the conversation ends.

---

4. Technology Stack

Component	Technology
Backend Framework	FastAPI
Database	PostgreSQL
Cache	Redis
AI Runtime	Ollama([1])
Frontend	React
Programming Language	Python 3.11

5. System Workflow

5.1 Per-Turn Processing

1. **User sends a prompt to the copilot.**
2. **PIHS receives the request and stores it.**
3. **PGCP resolves the user's identity and permissions.**
4. **The AI Engine sends the prompt to the SLMs in parallel.**
5. **Each model returns a governance analysis.**
6. **The Turn Judge combines the results into one rationalized output.**
7. **Results are stored in PostgreSQL.**
8. **Runtime metrics are updated.**

5.2 Session Finalization

When the session ends:

1. **PIHS waits for all background tasks to complete.**
2. **The Session Judge analyzes all turns together.**
3. **A session-level summary is created.**
4. **Results are displayed on the forensic dashboard.**

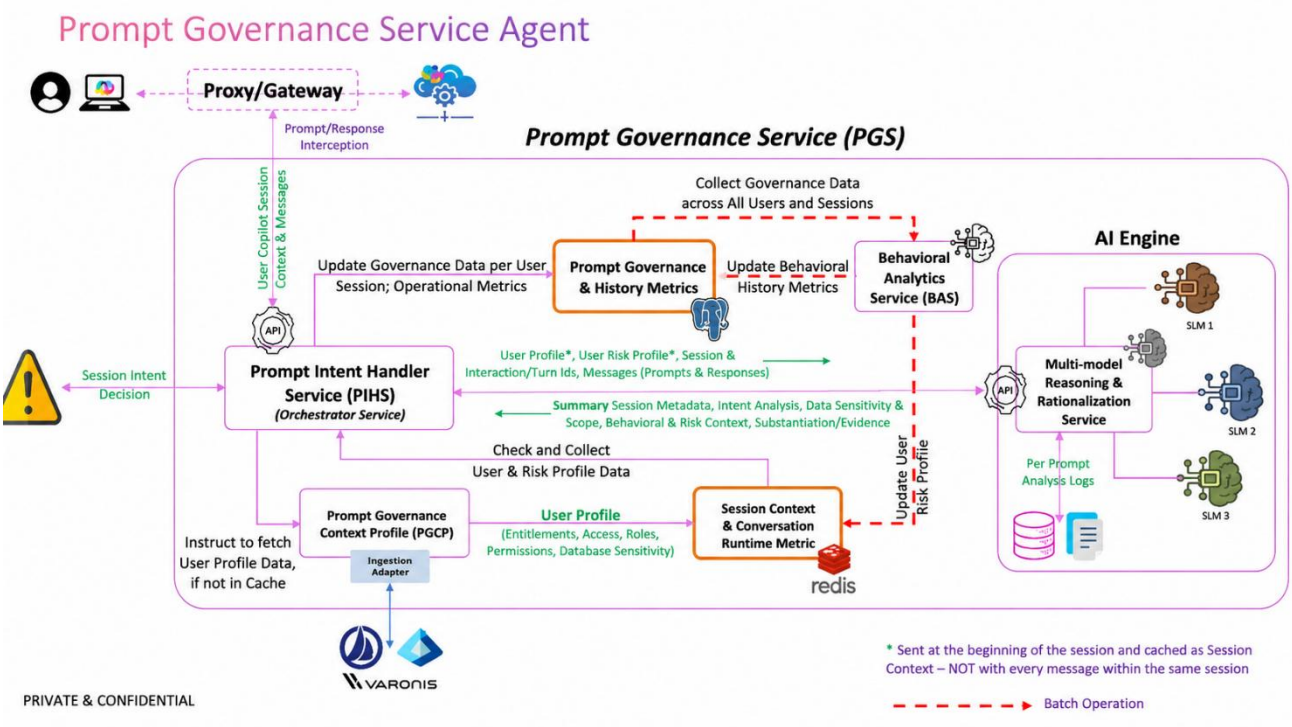

Fig 2. Internal architecture of PGIH

6. Key Features

6.1 Parallel AI Analysis

Multiple language models analyze each prompt at the same time. This improves reliability and reduces the chance of one model making incorrect decisions.

6.2 Outlier Suppression

One model sometimes produced unusually high risk scores for harmless prompts. To fix this, the AI Engine compares model outputs and suppresses obvious outliers before sending results to the judge model.

6.3 Identity-Based Risk Scoring

Risk is evaluated using the user's actual role and permissions.

For example:

- A financial analyst accessing vendor data may be low risk.
- A help desk employee making the same request may be high risk.

6.4 Session-Level Behavioral Tracking

The system tracks behavior across multiple prompts, not just one prompt at a time.

This helps detect:

- Enumeration attempts
- Escalating risk behavior
- Data exfiltration patterns

---

7. Testing and Validation

Testing was done manually using curl requests and log analysis.

Example test:

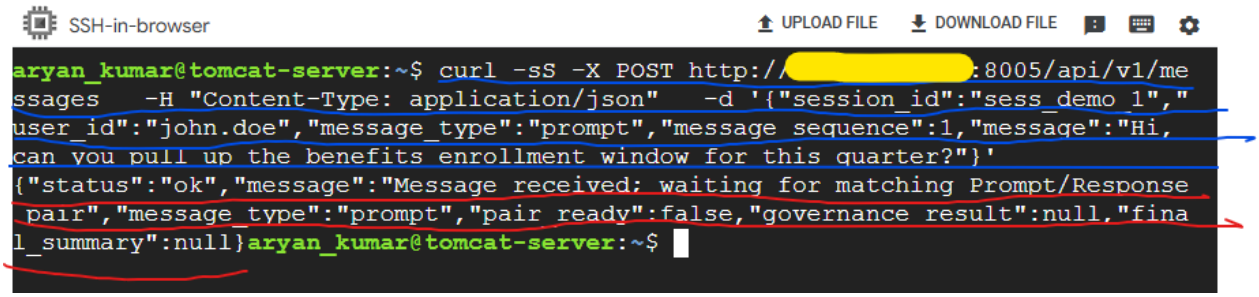

Fig 3: Testing out each turn

The following were checked during testing:

- Schema consistency
- Risk score accuracy
- Session tracking
- Multi-turn stability
- Judge output quality

---

8. Results

After improvements to prompts, outlier suppression, and consensus logic, the system produced more accurate governance results.

Improvements achieved:

- Better risk calibration
- Stable multi-turn outputs
- Reduced false positives
- Better session-level analysis

Typical processing time per turn was around 50–80 seconds, but users were never blocked because all analysis ran in the background.

Model	Runs	Cases	JSON Valid %	Latency Avg(s)	Stability %	Risk Score	Overall Score
gemma:2b-instruct	30	10	100.0	0.88	100.0	2.11	100.0
mistral:7b	30	10	100.0	3.01	100.0	24.14	99.75
gemma:2b	30	10	100.0	0.97	100.0	17.00	99.4
qwen2:7b	30	10	100.0	3.12	90.0	25.88	95.73
phi4-mini:latest	30	10	100.0	1.02	90.0	28.54	95.66
llama3.1:8b	30	10	100.0	3.21	80.0	19.15	91.71
phi4:latest	30	10	100.0	5.44	90.0	26.97	88.85
gemma2:2b	9	3	100.0	10.67	66.67	25.1	84.62
phi3:latest	30	10	60.0	1.97	90.0	35.58	80.0
gemma:7b-instruct	30	10	100.0	2.15	90.0	4.53	76.0
gemma2:27b	9	3	100.0	86.84	33.33	28.04	53.33
deepseek-r1:7b	9	3	100.0	65.62	0.0	25.74	45.0

Fig 4: Testing and finalizing SLM's to use for backend

---

9. Engineering Challenges

9.1 Model Drift

Problem

The language models started responding like chat assistants instead of governance auditors during long sessions.

Solution

The session history was simplified so the models only saw governance metadata instead of full conversations.

9.2 Risk Inflation

Problem

One model consistently marked normal financial prompts as medium risk.

Solution

Outlier suppression logic was added before the judge step to get more accurate results.

9.3 Broken Log Files

Problem

Deleting log files while services were running caused logs to disappear permanently.

Solution

WatchedFileHandler was used so log files automatically recreated themselves.

---

10. Future Work

Possible future improvements include:

- Automated testing with pytest
- Fine-tuned governance-specific models
- Faster streaming inference
- Active blocking of dangerous AI responses
- Cross-session anomaly detection
- Better evaluation datasets

---

11. Conclusion

The PGIH platform demonstrates that AI-powered governance for enterprise copilots is possible using locally hosted language models and microservices.

The project successfully combines:

- Parallel AI inference
- Identity-aware governance
- Session-level behavior tracking
- Risk analysis
- Distributed system design

PGIH provides a strong foundation for future enterprise AI governance systems and shows how organizations can monitor AI usage without disrupting employee workflows.

Part 2

Name: Aryan Kumar
Duration: May 26, 2026 – Aug 4, 2026
Supervisor Name: Shambhu Upadhyaya

---

12. Phase II Overview and Motivation

Part 1 (PGIH) focused on **monitoring** and understanding user intent and scoring risk while a person interacts with a copilot. Part 2 extends the project from monitoring toward **active data protection** in **agent-to-agent (A2A)** communication, where autonomous AI agents exchange data with one another on a user's behalf.

As enterprises adopt multi-agent systems, sensitive information (Social Security numbers, driver's licenses, passports, financial identifiers) is passed between agents that each have a different, legitimate need to know. For example, a background-check orchestrator may need to send a candidate's record to both a credit-bureau agent and a driving-history agent but each agent should only see, in cleartext, the specific fields its task actually requires.

The goal of Phase 2 was to build a system that could, for every agent-to-agent message:

- Intercept the payload before it reaches the recipient agent
- Reason, per recipient and per task, about which data may remain in cleartext
- Instruct an encryption service to protect everything else
- Verify the result and fail closed if any sensitive value was exposed
- Preserve the message structure so the recipient agent still functions

This is built from two cooperating components: **AURA** (the context and reasoning engine, which was my focus) and **AutoPilot** (the classification and encryption engine that runs on hard specified rules).

13. System Architecture

I inserted a governance and protection step into the agent-to-agent path. A customer proxy intercepts an agent's outgoing message and calls AURA. AURA reasons over policy, instructs AutoPilot to encrypt, verifies the protected payload, and returns it for delivery.

Flow: Orchestrator / Proxy → AURA (reason) → AutoPilot (encrypt) → AURA (verify) → Recipient Agent

AURA - Context and Reasoning Engine. AURA intercepts the payload, reasons over the recipient agent's Secure Agent Card and the task, decides which data types may remain in cleartext, instructs AutoPilot, verifies the outcome, and returns the protected payload. A key design principle is that **AURA never encrypts data itself**, it only decides and verifies. This keeps the reasoning layer independent of the encryption implementation.

AutoPilot - Classification and Encryption Engine. AutoPilot performs value-based PII detection and encryption. It exposes a **Rules-Based Exclusion (RBE)** interface: by default it encrypts every sensitive value it detects, and AURA supplies a list of field-type tokens to keep in the clear.

---

14. Secure Agent Cards and the Policy Model

Every recipient agent is described by a **Secure Agent Card**, a JSON policy document that declares what data the agent may receive in cleartext, what must always stay protected, and which agents it may communicate with. AURA reasons against the card of whichever agent a message is bound for.

Recipient Agent	Allowed in Cleartext	Always Protected
Credit-Bureau Agent	name, ssn, date of birth, address	driver's license, passport, email, phone
Driving-History Agent	name, driver's license, issuing state, dob	ssn, address, passport, email, phone
Criminal-History Agent	name, dob	ssn, driver's license, address, passport

Because the policy depends on the recipient, the **same candidate record produces a different protected payload for each agent**.

15. AURA Reasoning Engine and Workflow

For each intercepted message, AURA runs the following pipeline:

1. **Parse** the candidate payload out of the message envelope.
2. **Classify** every value into a PII type (see Section 16).
3. **Decide**, using a locally hosted language model, which optional types this agent may keep in the clear while enforcing hard policy floors where required-clear fields are always clear, and must-protect / payment-card fields are always encrypted, regardless of the model's opinion.
4. **Instruct** AutoPilot with the exclusion token list.
5. **Verify** the returned payload and fail closed on any leak (see Section 18).
6. **Return** the protected payload to the proxy for delivery.

The language-model step is what makes protection **context-aware**. AURA produces a short, human-readable reason for each decision (for example, *"the credit bureau agent needs the address to verify the applicant's location, but other identifiers are not necessary for this task"*) instead of applying a fixed field list.

---

16. Type-Based Classification of Nested Payloads

Real candidate records are deeply nested and use generic key names, so sensitive values do not appear under predictable field names. A driver's license number may sit at drivers_license.number, and a reference's phone number at references[].phone.

To handle this reliably, AURA classifies by **canonical type** rather than by field name. It walks the payload recursively and assigns each value a type using **parent-aware** rules — for example, drivers_license.number becomes the *driver_license* type, current_address.state becomes *address*, and first_name / last_name become *name*. Because the decision is made on types, protection stays correct no matter how the data is nested or named.

Result: deeply nested and third-party fields, including reference phone numbers and previous addresses, are detected and protected correctly, with the message's original structure fully preserved.

17. AutoPilot Integration (Rules-Based Exclusion)

AURA talks to AutoPilot through the RBE interface. The request is deliberately minimal of six fields, no redundant data:

Field	Purpose
original_path	Required by the AutoPilot API contract
pii_pointer_path	Where the payload lives in the body
fields_to_exclude	The keep-clear type tokens AURA computed
original_body	The candidate record to protect
original_method	Required by the AutoPilot API contract

AURA guarantees that fields_to_exclude only ever contains tokens from AutoPilot's official vocabulary, so the encryption engine never receives an unrecognized instruction. During integration I found that the free-text rules_text hint was **not honored** by AutoPilot and only made the request larger, so it was removed. The type tokens alone drive the exclusion, keeping every request lean.

---

18. Fail-Closed Verification

Because encryption is done by a separate service, AURA does not trust the result blindly. After AutoPilot responds, AURA independently confirms that every value it marked as strongly sensitive (SSN, card number, bank account, passport, driver's license, email, phone) has actually disappeared from the payload.

- If verification passes, the protected payload is returned.
- If a sensitive value survived, AURA retries; and if it still leaks, AURA **fails closed** which means it returns an error and delivers nothing, so a partially-protected record never reaches the recipient agent.

Design principle: the default outcome of any failure is non-delivery. Sensitive data is never forwarded on the assumption that it was protected.

19. Proxy Integration API

AURA exposes a single proxy-facing endpoint. The customer proxy forwards the same message envelope it already builds and names the task. AURA selects the matching agent card, protects the payload, and returns the result.

- **Request:** POST /aura/decide?task=credit_report (or driving_history), carrying the candidate record inside the standard message envelope.
- **Response:** a compact object with the decision (allow / deny / error), the reasoning, the exclusion tokens, and the fully protected payload ready to forward. The response was trimmed during Phase 2 to remove duplicated and internal-only fields, leaving only what the proxy needs.

---

20. Testing, Validation, and Results

The full workflow was validated end-to-end against the live proxy payloads, using a nested candidate record. Two things were checked: the **protection decision** (does each field end up clear or encrypted, per the agent's card) and the **protection integrity** (is every sensitive value actually encrypted, and is the structure preserved).

Protection Matrix

Field	Credit-Bureau Agent	Driving-History Agent
Name / Date of birth	clear	clear
SSN	clear	encrypted
Address (current & previous)	clear	encrypted
Driver's license	encrypted	clear
Passport / Email / Phone	encrypted	encrypted
Reference phone	encrypted	encrypted

Outcome: every sensitive value was protected exactly as each agent's card requires, all fields were handled correctly, and the message structure was preserved in every case. Sensitive-data leaks across all tests were zero.

End-to-End Latency (measured from an external client over the network):

Stage	Credit Report	Driving History
AURA reasoning + verification	~1.7 – 2.5 s	~1.7 s
AutoPilot encryption	~0.9 s	~0.9 s
Total (end-to-end)	**~2.5 – 3.5 s**	**~2.5 s**

The reasoning step dominates the latency and is the main target for future optimization.

21. Challenges (Part 2)

21.1 Response Corruption from Echoed Rules Text

Problem: AutoPilot appended a text hint to its JSON response, producing invalid JSON. This made AURA treat correctly-encrypted responses as unusable and repeatedly fail closed.

Solution: AURA now parses only the leading JSON object of the response and ignores any trailing text, and the unnecessary hint was removed from the request entirely. The fault was isolated through a controlled comparison that proved AURA's request was byte-identical to a working one, which narrowed the problem down to response parsing.

21.2 Substring Encryption of Embedded State Codes

Problem: AutoPilot encrypts a detected value wherever its characters literally appear. A synthetic license like OH-DL-82736491 had only its OH substring encrypted, corrupting the value and exposing the remaining digits.

Finding: Realistic license formats (which contain no embedded state code) are handled correctly encrypted for the credit agent and kept clear for the driving agent. The issue was documented and reported to the AutoPilot team as an edge case affecting only malformed identifier formats.

21.3 Inconsistent Token Handling

Problem: Certain exclusion tokens were honored inconsistently inside full records.

Solution: AURA limits its fail-closed verification to genuinely sensitive, reliably-detectable values, accepts harmless over-encryption without failing the request, and reports the remaining gaps to the encryption team instead of working around them incorrectly.

---

22. Discussion

Applicability and Adoption

The research and prototype are designed to be adopted incrementally rather than as an all-or-nothing platform. An organization deploying AI can use PGIH as a governance and monitoring layer to understand how employees use AI, while a team building multi-agent systems can use AURA to enforce least-privilege data sharing between agents. Because the system is built as independent microservices communicating over HTTP, each component can be adopted on its own and connected to existing infrastructure through a proxy and a single API, keeping integration effort low. A further practical advantage is that all reasoning runs on locally hosted models, so the approach can be deployed in privacy-sensitive environments such as healthcare, finance, government without sending any data to a third-party service. For researchers, the more reusable contributions are the design patterns themselves: type-based classification of sensitive data in nested payloads and context-driven per-recipient policy decisions These patterns are not tied to this particular implementation and can be applied to other governance or data-protection systems.

Model Evaluation and Justification

Selecting the language model used in the backend was one of the central engineering decisions, because the model drives both the risk reasoning in Part 1 and the protection decisions in Part 2. Several small language models including Mistral 7B [1], Llama 3 [2], and Gemma [3] were evaluated by running the same governance prompts through each and comparing them on a consistent set of criteria: whether the output followed the required schema, how accurately and consistently the risk was scored, how stable the reasoning remained across multi-turn sessions, the quality of the explanations produced, and the model's reliability and latency on the available hardware. Models that drifted into chat-assistant behavior during long sessions, inflated the risk of ordinary requests, or produced malformed output were set aside. Mistral 7B was chosen because it produced the most consistent, schema-valid output with stable reasoning and no runtime problems on CPU and GPU hardware. It is important to be precise about what "satisfactory" means here: the model reliably met the two hard requirements of the service which was correct schema and correct detection of sensitive data while keeping risk scoring reasonable. This evaluation was comparative rather than an exhaustive, benchmarked study, so the honest claim is that Mistral 7B is one of the strongest choices for this use case and hardware, not that it is provably optimal. A formal benchmark against a labeled evaluation dataset is identified as future work and would let the selection be stated with quantitative confidence.

Scalability and Path to Production

The current prototype runs comfortably for demonstration and testing, and its main performance cost is language-model on CPU and GPU, which accounts for most of the roughly two to three seconds of per-request latency. The architecture was, however, designed with scaling in mind. The services are stateless and all session and audit state lives in Redis and PostgreSQL so the AI Engine can be replicated horizontally behind a load balancer to handle more concurrent traffic. Latency can be reduced substantially by moving inference to better GPU only hardware or to smaller, quantized models, and by caching decisions for repeating agent-and-task pairs so that identical requests skip the model step entirely. All input/output is already asynchronous, which helps throughput under load. Reaching a production deployment would primarily be an infrastructure exercise rather than a redesign: containerizing the services and running them under an orchestrator such as Kubernetes, adding GPU nodes and a dedicated model-serving layer, using managed and replicated database and cache tiers, and adding the operational necessities of rate limiting, monitoring, and autoscaling. Because the system was built as loosely coupled microservices with external state and asynchronous communication from the start, scaling it is a matter of adding capacity and hardening operations, not rebuilding the core features.

---

23. Future Work and Overall Conclusion

Future Work

- **Decision caching** - reuse the reasoning result for repeated agent/task pairs to skip the language-model step and cut latency to just the encryption call.
- **Accelerated inference** - move the reasoning model to a GPU or a lighter model for faster, more consistent latency.

- **Corruption guard** - automatically detect partially-encrypted fields and redact or fail closed.
- **Broader policy coverage** - more Secure Agent Cards and richer per-field policy rules.

Overall Conclusion

Across both phases, this project progressed from **observing** AI usage by users to **actively protecting** the data that AI agents exchange. Part 1 (PGIH) established identity-aware, session-level governance of human–copilot interactions. Part 2 (AURA) extended that foundation into agent-to-agent communication, adding context-driven, per-recipient data protection with independent, fail-closed verification. Together they show a complete governance lifecycle understanding intent and risk, then enforcing least-privilege data exposure using locally hosted language models and a microservice architecture, without disrupting the systems they protect.

---

24. References

[1] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. "Mistral 7B." arXiv:2310.06825, 2023.

[2] Llama Team, AI @ Meta. "The Llama 3 Herd of Models." arXiv:2407.21783, 2024.

[3] Gemma Team, Google DeepMind. "Gemma: Open Models Based on Gemini Research and Technology." arXiv:2403.08295, 2024.

[4] OWASP Foundation. "OWASP Top 10 for Large Language Model Applications." 2025.

[5] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection." arXiv:2302.12173, 2023.

[6] National Institute of Standards and Technology. "Artificial Intelligence Risk Management Framework (AI RMF 1.0)." NIST AI 100-1, 2023.

Signature:
Aryan Kumar
8/10/2026