

**Multi-Agent System in Digital Forensics**

Transcript for the

team project presentation

Provided by: Group 4

# Slide 1: Title

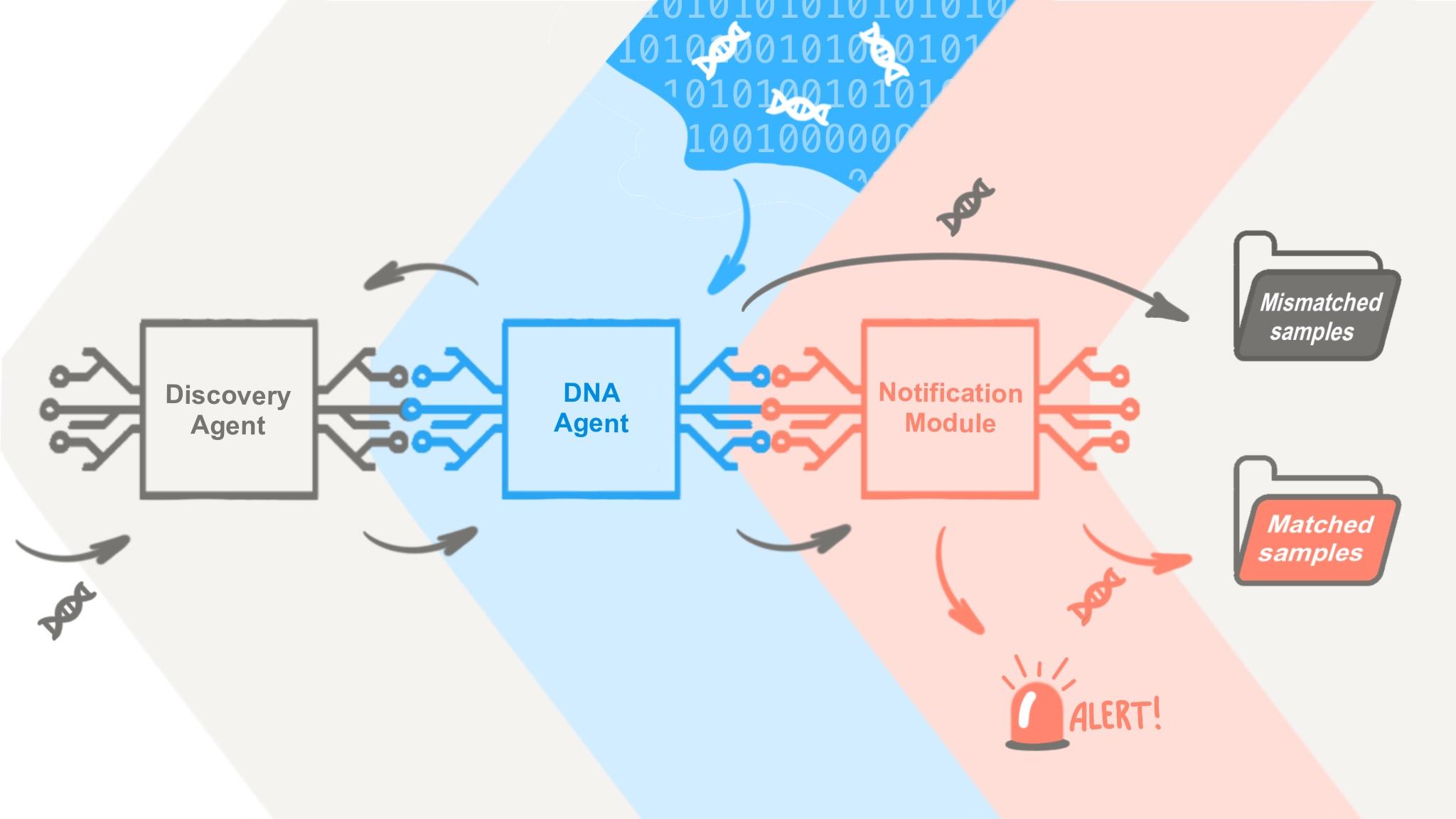
# Slide 2: Introduction (Vasilisa)

Hello everyone, we are going to present a system for the Digital Forensics domain, designed for use by a detective agency. Its purpose is to locate specific file types within a file system, archive them, and send the results to a designated location for further analysis. We propose the implementation of a multi-agent system to accomplish this task.

A multi-agent system (MAS) is a system in which independent software components work collaboratively to solve problems that cannot be addressed by a single agent. The key characteristics of intelligent agents include autonomy, reactivity, proactivity, adaptability, as well as communication and coordination with other agents (Kendrick et al., 2016).

In this instance, our objective is to locate a DNA match for potential serial killer suspects by comparing their DNA samples with a comprehensive DNA data lake. For example, our aim is to determine if the DNA matches those recovered from victims within a specific area over the past 20 years, utilising the database that contains cases yet to be investigated.

# Slide 3: The general idea of the architecture (Vasilisa)



The system requires inputs in a specific format for the digital DNA samples. We have identified two main agents for the system, according to two main tasks - to obtain a sample for the analysis and to compare it with the samples in the database. Right now we present 2 agents for simplicity to demonstrate the key ideas of the system, however it is possible to add more agents in future.

First agent is a Discovery agent (Agent 1): This agent is responsible for searching for new pieces of information to compare them with the database further. It keeps track of the number of files that need analysis and instructs the next agent on which files to pre-analyse in order to identify suspicious files (Terra, 2023).

In our DNA example, the Discovery agent (Agent 1) is responsible for searching for new DNA samples to test. If new samples are found, they are sent to the DNA agent (Agent 2) for processing, and a response is expected.

The second agent is a DNA agent (Agent 2): This agent receives a DNA sample and checks whether the database contains a match for that specific DNA. If a match is found, the agent retrieves the case number associated with the DNA sample and includes both the DNA sample and the related case numbers in the package.

Each agent is equipped with an embedded analysis function, sensors to read from the queue, and actuators to perform actions or place responses in the queue.

The other team member, Alberto, has prepared an in-depth architecture, a class diagram, and a sequence diagram for the complex system currently under construction. He has appropriately emphasised the need for name-changing of the agents to accurately reflect their functionality and ensure consistency. And right now, he is ready to explain several key decisions regarding the project.

*Thank you, Vasilisa.*

# Slide 4: Development of the architecture (Alberto)

# We have four main entities that compose our architecture:

# The file system, where DNA file samples live.

# A DiscoveryAgent, responsible for fetching and sending DNA samples to the DNAAgent.

# A DNAAgent, responsible for verifying if a DNA sample was found in its internal database.

# A message queue used for communication.

# 

The principal reason to select a message queue for communication is to decouple agents and make the communication between them asynchronous. This approach follows a choreography pattern largely used in software engineering projects implementing microservice architectures. According to Tomar (2021) choreography assigns specific responsibilities to each actor in the architecture while ensuring actors are familiar with the overall plan.

Additionally, communication is performed using the Knowledge Query and Manipulation Language (KQML). Finin et al. (1993) state that KQML proves particularly valuable in facilitating communication among autonomous and asynchronous agent-based programs.

# Slide 5: Class diagram (Alberto)

# According to Maalal & Addou (2012) Three main steps for MAS designing are:

1. Maintain consistent titles for classes and associations within the diagrams.

2. Include each agent's roles, perceptions, intentions, and representations in the attribute section of the UML class.

3. Consolidate all methods or functions within the operations section of the UML class.

The main class is the Agent class, representing a basic intelligent agent: The agent percepts its environment via sensors, analyses data via its analysis module, and produces actions via its actuator module. Each agent has a name property agents use to send messages to each other.

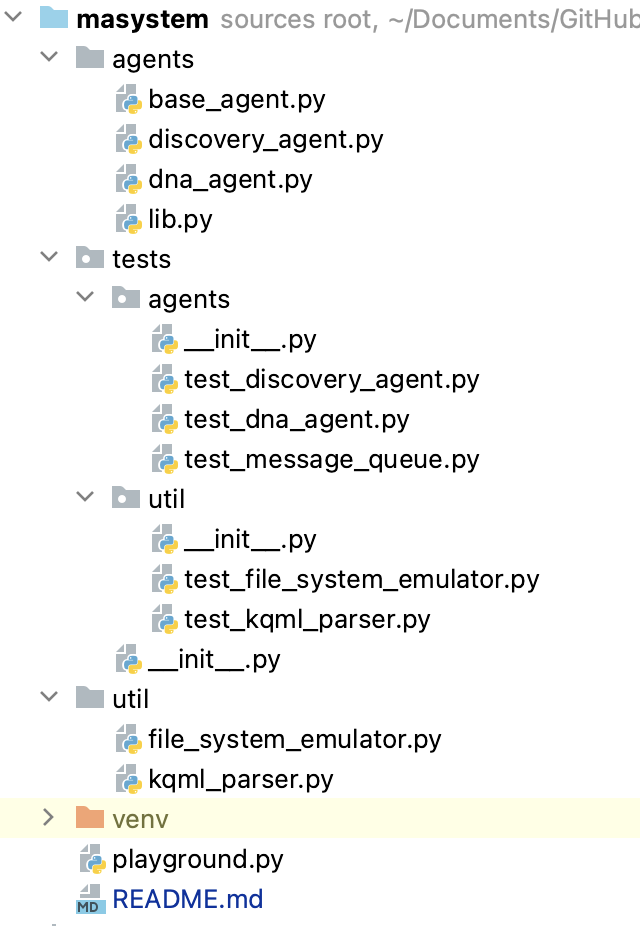
We have two types derived from the main Agent class: The DiscoveryAgent and the DNAAgent.

These two agents have different sensors, actuators, and analysis-module implementations. The discovery agent has a DiscoverySensor instance, a DiscoveryActuator instance, and a DiscoveryAnalysisModule.

On the other hand, the DNA agent has a DNASensor instance, a DNAActuator, and a DNAAnalysisModule.

Other classes are:

* The MessageQueue and Message class, which supports communication between agents.
* The CasesDatabase class, which helps with DNA sample storage.
* And the FileSystemEmulator class, which supports emulating the file system environment.

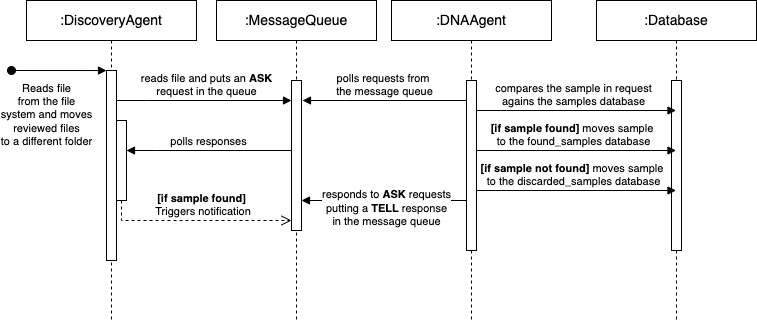


# Slide 6: Sequence diagram (Alberto)

The sequence diagram represents the following agents' cycle:

1. First, the DiscoveryAgent percepts its environment (the file system), reads a file to extract the DNA sample, and puts an ASK request into the message queue to ask if it is a match.
2. The Discovery Agent also checks if there are messages for it in the queue. These messages are responses from the DNAAgent. If the answer says a sample is a match, then DiscoveryAgent triggers a notification event.
3. The DNA agent polls the queue for ASK requests from DiscoveryAgent. When there's a request for a DNA sample review, the DNAAgent compares the sample against its internal database.
   1. If the sample is a match, the DNAAgent puts a TELL response in the queue saying the sample exists in the database and inserts the sample in the found-samples-database.
   2. Otherwise, the DNAAgent puts a TELL response saying the sample doesn't exist in the database and inserts the sample in the discarded-samples-database.

This cycle can repeat indefinitely. Now, Maksym will describe in more in-depth the implementation details of our agents.



# Slide 7: Class Agent code example (Max)

On slide 7, there is a basic code for abstraction over the Agent. The Agent consists of three modules: sensor, actuator, and analysis module. Sensors perceive the environment and obtain data for further processing. The analysis module analyses the data and determines the actions the agent needs to take. The actuator provides the business logic for executing particular actions.

The Agent class has one main method called "loop" which sequentially runs three methods from the provided modules. This architecture is simple and universal, allowing agents to be built using prepared and tested blocks. The business logic for the modules is injected into the Agent object through the constructor method.

To create an agent, it is only necessary to instantiate a new object and initialise the required modules. The slide provides an example of initialising the discoveryAgent.

This approach enables the use of a sophisticated and flexible code organisation technique called dependency injection. In the example code, we use a simple dependency injection approach for better transparency. However, for larger systems, more sophisticated frameworks can be utilised. The structure of the system can be described in configuration files (XML, YAML) in a highly readable format. The system kernel can then automatically create a working prototype using the building blocks described in the configuration.

# Slide 8: Class Message code example (Max)

In our multi-agent system architecture, we have identified another important aspect, which is highlighted on slide 8. We introduced two classes, MessageQueue and Message, to handle agents' communication.

The Message class defines the structure of messages exchanged between agents. We decided to use separated class because using its type specifications in all interfaces allows us to enforce communication formats, leading to a more stable system. While this approach may introduce some rigidity and complexity to the design of Agent modules, it provides greater stability and predictable behavior, especially in larger systems. Additionally, having a wrapper around raw text messages enables us to incorporate validation, logging, and other useful functionalities.

The MessageQueue serves as a simple queue manager, representing a basic list with operations like append, which adds new messages to the queue, and pop, which retrieves messages addressed to specific agents. We designed the MessageQueue as a Singleton to mimic other shared state queues, such as databases or sophisticated queue managers like RabbitMQ. In our test code, the queue's state is shared among agent objects. However, it can easily be refactored to utilize any external queue messenger.

For any module requiring access to the shared message queue, it simply needs to instantiate a new instance of the MessageQueue class. Since it is a Singleton, the instance is shared among all objects within a script, providing consistent communication throughout the system.

# Slide 9: Testing the code (Max)

On the slide 9 there is a test code for our system. We decided to create a playground where all the components of the system are works in a simple manner. The DNA database is represented by simple class CasesDatabase, which provides basic functionality for search and adding new data. Both agents are represented in test code as discoveryAgent and dnaAgent.   
Cases requests are generated randomly by FsEnvironment class by using emulator FileSystemEmulator. It generates text files with simplified DNA samples. Also, a new instance of MessageQueue is created, to represent shared channel for communication.   
Systems works in eternal loop. On each iteration environment and agent do some activity, which produced some logs record in output. Logging is simplified to show only key aspects of multi-agent system workflow.

Agents 1 and 2 utilise MessageQueue as a shared place for exchanging messages for agents, and we use agent communication language (ACL) format to unify knowledge transfer through the messages. ACL describes message structure, MessageQueue makes message sending possible.

MessageQueue is not an ACL. It is not specifically designed as an agent communication language, however it can be used as a communication mechanism within a multi-agent system. MessageQueue is not inherently worse than an agent communication language like KQML. The suitability of MessageQueue versus an ACL like KQML depends on the specific requirements and context of the system being developed.

MessageQueue, in and of itself, does not offer direct support for negotiation, cooperation, collaboration, and coordination between agents. Typically, agent communication languages (ACLs) are used for these purposes. However, in our detective case, we actually do not want to enable agents to engage in negotiation deeply.

We used this combination, because it is suitable for the proposed task, providing a simple and efficient way to send and receive messages between components or applications.

Slide 10: Conclusion (Vasilisa)

Thank you, dear colleagues. Data privacy and security pose significant challenges in the development of applications that utilise sensitive data, such as biometric or medical data. In our current case, we are using simulated data; however, in real-world scenarios, it becomes crucial to employ cryptographic techniques to ensure the safety of sensitive information.

Moreover, the development of intelligent agents raises ethical concerns. According to Leslie (2019) and his FAST principles, systems involving intelligent agents must prioritise transparency. This implies that these systems should be explainable to both users and stakeholders.

In conclusion, from an architecture perspective all mentioned challenges also affect the way a multi-agent system should be designed.   
  
The implementation of the multi-agent system utilised a simple and universal Agent class to define the basic structure of agents within the system. This approach enabled us to encapsulate the complexity of the system's business logic within its modules (Bergenti & Poggi, 2000). By developing the system using separate and well-tested building blocks, we were able to compose a system that could be easily modified and extended. This approach also allowed us to involve domain specialists with specific expertise in developing complex aspects of the system's business logic and leverage modern testing approaches.

The introduction of a centralised messaging queue played a crucial role in the system's implementation. It provided a universal communication format and enhanced the security of the communication channel. Furthermore, the use of ACL implementation formalised the communication between agents and facilitated the transfer of complex data throughout the system (Soon et al., 2019)

Despite these advantages, the proposed architecture does have significant drawbacks. The modular approach, although beneficial for individual module development, adds complexity to the composition of the final system. Similar to microservice architecture, additional documentation on the architecture of agent communication becomes necessary. Any mistakes in this communication process can have a significant impact on the entire system. Therefore, it is crucial to provide comprehensive documentation and ensure careful planning and execution when composing agents within the system. Thank you for listening!

# References

Bergenti, F. & Poggi, A. (2000) Exploiting UML in the design of multi-agent systems. In *Engineering Societies in the Agents World: First International Workshop, ESAW 2000 Berlin, Germany, August 21, 2000 Revised Papers 1* (pp. 106-113). Springer Berlin Heidelberg.

Bianchi, L. & Liò, P. (2007) Forensic DNA and bioinformatics. *Briefings in bioinformatics*, 8(2), pp.117-128.

Finin, T., Fritzson, R., McKay, D. & McEntire, R. (1994) KQML as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management* (pp. 456-463).

Kendrick, P., Hussain, A.J. & Criado, N. (2016) Multi-agent systems for dynamic forensic investigation. In *Intelligent Computing Theories and Application: 12th International Conference, ICIC 2016, Lanzhou, China, August 2-5, 2016, Proceedings, Part I 12* (pp. 796-807). Springer International Publishing.

Leslie, D. (2019). Understanding artificial intelligence ethics and safety. The Alan Turing Institute. <https://doi.org/10.5281/zenodo.3240529>

Maalal, S. & Addou, M. (2012) A new approach of designing Multi-Agent Systems. *arXiv preprint arXiv:1204.1581.*

Mahdi, G., Gouaïch, A. & Michel, F., 2010, May. Real-time Coordination Of Multi-Agent Systems: An Integrated View. In *AAMAS'10: Agents in Real-time and Dynamic Environments Workshop* (pp. N-A).

Nahm, Y.E. & Ishikawa, H. (2005) A hybrid multi-agent system architecture for enterprise integration using computer networks. *Robotics and Computer-Integrated Manufacturing, 21*(3), pp.217-234.

Shtrauss, A. (2023) Biopython | Bioinformatics Basics. Available from: <https://www.kaggle.com/code/shtrausslearning/biopython-bioinformatics-basics> [Accessed 8 Jun 2023]

Soon, G.K., On, C.K., Anthony, P. & Hamdan, A.R. (2019) A review on agent communication language. *Computational Science and Technology: 5th ICCST 2018, Kota Kinabalu, Malaysia*, 29-30 August 2018, pp.481-491.

Terra, J. (2023) Exploring Intelligent Agents in Artificial Intelligence. AI & Machine Learning Course on SimpliLearn. Available from: <https://www.simplilearn.com/what-is-intelligent-agen-in-ai-types-function-article> [Accessed 10 Jun 2023]

Tomar, A. (2021) Microservices Orchestration vs Choreography | (Technology). Available from <https://medium.com/geekculture/microservices-orchestration-vs-choreography-technology-5dbe612cf7e9> [Accessed on 14 Jul 2023]

Twardowski, B. & Ryzko, D. (2014) Multi-agent architecture for real-time big data processing. In 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) (Vol. 3, pp. 333-337). IEEE.