

UDC 004.8:37.018.46

Evolution of the neural network module architecture: supporting multimodal services in the Mirera digital educational platform

A. S. Karavaeva¹, A. M. Kulikov², N. S. Martynov³, K. A. Mashchenko⁴,
K. K. Pchelin⁵, E. D. Tarasuk⁶

¹National Research Center «Kurchatov Institute» - SRISA, Moscow, Russia, aleksandrakaravaeva@yandex.ru;

²National Research Center «Kurchatov Institute»- SRISA, Moscow, Russia, kulikrch@yandex.ru;

³National Research Center «Kurchatov Institute» - SRISA, Moscow, Russia, nikolai.martynov@math.msu.ru;

⁴National Research Center «Kurchatov Institute» - SRISA, Moscow, Russia; Lomonosov Moscow State University, Moscow, Russia;
State University of Management, Moscow, Russia, kirill.mashchenko@niisi.ru;

⁵National Research Center «Kurchatov Institute» - SRISA, Moscow, Russia, pcelinkosta0@gmail.com;

⁶National Research Center «Kurchatov Institute»- SRISA, Moscow, Russia, ekaterina.tarasuk@math.msu.ru;

Abstract. This paper examines the evolution of the neural network module architecture of the Mirera Digital Educational Platform in connection with the expanding range of applied artificial intelligence models. It is shown that the original LLM-oriented orchestrator core did not allow models of other modalities to be integrated without substantial reworking. A universal multimodal architecture is described, including a service manager abstraction based on a unified contract, configurable request-processing chains, and a unified stream of output artifacts. Examples of integrations are presented. It is shown that the proposed architecture enables the integration of new model types without modifying the router core and provides a basis for the further development of multimodal educational scenarios.

Keywords: digital educational platform, DEP Mirera, neural network orchestrator, multimodal models, microservice architecture, LLM, OCR, RabbitMQ, AI agents, educational technologies

1. Introduction

In recent years, large language models (LLMs) have become one of the key areas of practical implementation of artificial intelligence in education. The emergence of high-performance open architectures, their availability as ready-to-use API services, and the reduction of hardware infrastructure requirements through quantization methods have made it possible to move from isolated experiments to the systematic integration of language models into digital educational platforms. Today, LLMs are used for automated generation of learning assignments and test questions, review of learners' solutions, development of interactive assistants for specific disciplines, personalization of educational trajectories, and reduction of teachers' administrative workload [1]. At the same time, the range of associated artificial intelligence models is also expanding: optical recognition, image processing, speech recognition, and speech synthesis systems are being connected to language models, forming tools for working with multimodal scenarios in digital educational platforms.

The Mirera Digital Educational Platform [2] has accumulated experience in integrating artificial intelligence models into a wide range of educational tasks. In [3], the criteria for selecting large language models for educational scenarios are summarized, a comparative description of the architectures used in the platform is provided, and specific scenarios for their use are described: from generating assignments and test variants to building student and teacher assistants. In [4], the experience of integrating a neural network assistant for programming learners is presented; this assistant is capable of analyzing students' solutions in real time and generating instructional recommendations directly within the platform environment. In [5], a system for the automated generation of educational video materials based on a modular digital teacher twin is presented; it combines automatic speech recognition (ASR), text-to-speech synthesis (TTS), and lip synchronization (LipSync) modules and demonstrates a significant reduction in the teacher's labor costs for preparing educational video materials while maintaining their high quality. In addition to neural network services proper, the platform also develops related areas of educational

process automation, in particular hybrid attendance tracking with QR verification and video identification [6], which together form an integrated infrastructure for intelligent support of participants in the educational process.

The educational process is multimodal by nature: in addition to textual formulations of tasks and solutions, it involves images and diagrams from educational materials, audio and video recordings of lectures, handwritten notes, and teachers' presentations. As educational scenarios in a digital platform become more complex, the range of applied artificial intelligence models naturally goes beyond purely language-based systems: optical recognition systems for educational materials and handwritten text, speech recognition and synthesis models, and other neural network modules become increasingly relevant. Each of these areas requires models with their own input-data modality, specific hardware configuration (for example, specialized networks for optical recognition and vision-language models for documents with complex structure), and their own semantics of the output result, such as structured text, an audio signal, or a graphical representation. Implementing such scenarios within a single educational platform requires that the neural network module core be capable of connecting models of different natures without substantial reworking of routing, platform interaction contracts, or request-processing infrastructure.

The implementation of the scenarios described in [3; 4] required the creation of an independent neural network module that interacts with the Mirera platform core through message queues and performs the functions of request routing, model-pool management, and response assembly according to the contract with the platform. Initially, this module was designed for a single class of tasks: processing text requests using LLMs. This determined its architectural feature: all request-processing chains rigidly assumed that a language model stood at the end of the workflow, while attachment processing (documents and images) was performed locally inside the agent using a recognition library, thereby introducing substantial hardware dependencies into the monolith.

With the emergence of tasks involving other modalities, the incompatibility of this architecture with an expanded range of models became apparent. Connecting each new type of service within this architecture would require either duplicating the routing infrastructure or making point modifications to the core for the specifics of a particular service; both approaches lead to increasing complexity and poorer maintainability. In addition, placing recognition

dependencies inside the monolith did not allow resources for recognition tasks and LLM inference to be scaled independently, although they have significantly different load profiles and hardware requirements. By LLM inference, we mean the stage at which an already trained LLM processes a user request and generates a response without changing its internal parameters.

This paper describes the architectural evolution of the neural network module of the Mirera platform toward universal support for multimodal services. Section 2 examines the original LLM-oriented architecture and its limitations. Section 3 describes the transition to a universal multimodal architecture with a unified service manager contract, configurable chains, and a unified stream of output artifacts. Section 4 presents practical implementations of the capabilities of the new architecture.

2. Original LLM-Oriented Architecture

2.1 Two-Phase Request Processing

The neural network module of the Mirera platform is based on asynchronous request processing through the RabbitMQ message broker, which uses the AMQP protocol [7]. Requests from the platform arrive as structured envelope messages in an intake queue; responses are placed in a notification queue and delivered back to the platform for displaying the result to participants in the educational process. Between these endpoints, request processing is divided into two phases, each implemented by its own message-consumer loop.

In the intake phase (Intake loop), the router validates the structure of the incoming envelope message, loads the description of the processing chain for the corresponding command from a configuration file, checks the payload for compliance with the command contract, and places the message in the queue of the first service in the chain. This phase does not perform computational work and provides strict input control: invalid requests with a malformed message structure, an unknown command name, or a payload that does not comply with the contract are rejected before they reach load-sensitive models. Due to the absence of heavy dependencies, the intake loop can be scaled independently of execution loops, which is especially important during peak platform loads, such as the beginning of classes, mass submission of student solutions, and assessment periods.

In the execution phase (Service loop), the service manager retrieves a message from the queue of the corresponding service, transforms the input fields according to the description of the chain step,

validates the request against the strict schema of a specific pipeline, performs model inference, and then transforms the output fields into artifacts. If the chain contains a next step, the message is placed in the queue of the next service; if the step is the last one, the response is assembled according to the command contract and sent to the notification queue of the Mirera platform. Each service manager keeps its own pool of models in a ready state in memory and acknowledges processing of the next message to the broker only after successful completion of the corresponding step. This ensures delivery atomicity and system resilience to failures of individual components: if processing terminates abnormally, the message is returned to the queue and picked up by another service-manager instance.

2.2 LLMServiceManager as a Single Point of Model Interaction

Previously, only one service-manager class, LLMServiceManager, was registered in the router registry. This component performed several responsibilities: managing a pool of language-model instances, resolving concurrent access conflicts through a model acquisition and release mechanism, selecting a specific pipeline by name, and directly interacting with the model through the LLManager subsystem. The model pool was maintained in a hot-standby mode: when the service started, all models used by it were loaded into the video memory of graphics accelerators. This eliminated reinitialization delays, ranging from tens of seconds to several minutes per model depending on its size and loading method, and provided low response latency when a request arrived from the platform.

In addition to model support, the service manager performed the functions of a primitive load balancer between instances of the same model: when a request arrived, it searched for a free instance among the busy ones and, only if none was available, returned a temporary-unavailability flag prompting the broker to requeue the message after a short delay. LLMServiceManager also encapsulated logic for polling model state and collecting application metrics, including counters of processed requests, response-time histograms, and inference-error markers.

A separate pipeline was defined for each educational task: a self-contained module comprising a strict request and response schema, a prompt template, and execution logic. The pipeline entry point was an asynchronous function that accepted a validated request, a model instance, and a reference to the service manager; the returned value was a dictionary of named artifacts. Pipeline lookup was organized in two levels: first, a private folder with implementations specific to the Mirera

platform was checked, and then a common folder with generalized pipelines was checked. This organization made it possible to reuse common pipelines and, when necessary, override them with private versions without changing the router core, thereby simplifying functional development and ensuring continuity between generations of implementations.

2.3 Local Processing of Attachments

Educational tasks in the Mirera platform involve working with a variety of attached materials: texts of assignment statements and student solutions in PDF and DOCX formats, teachers' presentations in PPTX, images containing handwritten solutions to mathematical and engineering problems, and scanned copies of notes. In the original architecture, these attachments were processed locally inside the agent: the ContentProcessor component combined several document-parsing libraries and provided pipelines with already prepared text in a unified form. A central role among these libraries was played by Docling [8], a specialized IBM system combining the PP-DocLayoutV3 neural network page-layout segmentation model and a network component for extracting structured text; it is capable of processing complex documents with tables, formulas, and multi-column layouts.

This approach ensured the simplicity of the initial implementation and a unified interface for pipelines, but it quickly revealed a number of disadvantages. The network component of the Docling library required graphics accelerator support, which significantly increased the agent initialization time: in a standard configuration, startup took several tens of seconds because segmentation models had to be loaded into video memory. This reduced deployment flexibility: even in scenarios in which attachment processing was not expected, the agent still reserved graphics-accelerator resources for the recognition library. Any modification or update of the parser required a full rebuild and restart of the monolith, whereas changing the version of a language model should generally not affect attachment processing, and vice versa. Finally, it was impossible to scale resources independently for recognition tasks and language-model inference: as the flow of attachments increased, graphics accelerators had to be added specifically to the agent, even though there might be no load on the language models at that time, and vice versa.

2.4 Limitations of the Original Architecture

An analysis of the operation of the described architecture made it possible to identify three key limitations that hindered the integration of models

of other modalities and the development of the system as a whole.

First, the processing chains and response generation rigidly assumed that a language model was placed at the end of the pipeline and that its output message was text of a certain structure. For each command, the router code contained a specific wrapper that extracted individual fields from the LLM result and assembled the response for the platform. This violated the principle of separation of concerns: the router core code became dependent on the details of specific pipelines, and any change in the response format of a specific educational task required modifications to the core. A model of another modality, for example an optical recognition system returning document markup, would require the addition of its own wrapper variant and conditional logic for selecting the response assembly method depending on the type of service, which would only further increase the existing code coupling.

Second, only one service was registered in the service-factory registry, and the interface between the router and the service manager was not separated from the specifics of language models: its signatures contained concepts meaningful only for LLMs, such as generation parameters, context-window length, and system-prompt handling. Adding a new service type would require either repeating this specificity in a new class with dummy values for fields that have no meaning for models of another modality, or performing a deeper reworking of the router and related interfaces with changes propagated to all existing pipelines.

Third, incorporating document processing into the code of the agent itself created a rigid coupling between resource planning for the neural network module and for document recognition, as described in Section 2.3. This was compounded by organizational complexity: updating recognition components required coordination between developers of the neural network module and specialists maintaining the recognition infrastructure, even though these subsystems conceptually solve independent tasks.

Taken together, these limitations made the development of the platform toward multimodal request processing impractical without substantial architectural reworking. The next section describes the principles underlying the new architecture that eliminates these limitations.

3. Universal Multimodal Architecture

3.1 Principle of Separating Routing from Model Operation

The redesigned architecture is based on the

principle of strict separation of responsibilities between the request router and the service managers of specific models. The router is responsible only for receiving messages, validating their structure, routing them through service queues, and assembling the response according to the command contract; it contains no information about which model types are connected to the platform or how they operate. Service managers, in turn, encapsulate all specifics of working with models of their type, from managing a set of instances to formatting input and output data.

The connecting element between these components is the `BaseServiceManager` class, which defines a unified service-manager contract through four mandatory asynchronous methods: a method for finding a free model instance for the specified pipeline, a method for reserving a model for exclusive request processing, a method for releasing a model after processing is complete, and a method for executing the full “preprocessing, inference, postprocessing” chain within a specific pipeline. In addition, the class provides a default implementation of the application-metrics collection method, which ensures uniform monitoring of any connected service without the need to duplicate telemetry infrastructure in each derived class. Method signatures are formulated in terms neutral with respect to the model modality: instead of parameters specific to language models (context-window length, generation parameters, system-prompt handling), they use generalized concepts: pipeline identifier, request-field dictionary, and dictionary of named output artifacts.

Connecting a new model type is reduced to implementing this contract in a derived class and registering the service factory in the router registry. No changes to request intake, routing, error handling, or response assembly logic are required. The service-factory registry is a dictionary that maps a service name to the corresponding class; at startup, the router initializes one instance of each registered service and launches separate message-consumer loops for them on the corresponding queues. An additional practical effect of this separation is simplified unit testing: each service manager can be tested in isolation from the other system components using mock model implementations.

3.2 Configurable Processing Chains

The description of which services and pipelines a particular command passes through is moved from code to configuration files. Two files are defined for each command of the digital educational platform: a step-chain description and a contract for interaction with the Mirera platform.

The chain file specifies a sequence of steps with explicit indication of the service, pipeline name, and

rules for transforming fields between adjacent steps. Transformation may include renaming variables, reading values from S3 object storage at a specified path, and saving results back to S3. This is especially important for large artifacts, such as transcript texts, document images, and audio files, which are impractical to transmit in messages in their entirety because of broker message-size limitations. Paths in storage are specified relative to a common prefix associated with the request identifier, which ensures isolation of data from different requests and simplifies its disposal after processing is complete. An example of a complex chain is a command for transcribing and summarizing an audio recording of a class: the first step is executed in the speech recognition service and produces an output field with the transcribed text; the second step is executed in the language-model service with a summarization pipeline, for which this field is automatically renamed as an input field and supplied as context for generating a summary.

The contract file describes which fields the Mirera platform must provide in a request and how a successful response or an error response is assembled from chain artifacts. The router contains no response templates hard-coded in the source code: the entire semantics of the output message is determined by the contract, which makes it possible to add new commands and change the response format without modifying the core code. This is especially important for coordinating protocol versions with the Mirera platform: the evolution of message formats occurs at the configuration level and allows simultaneous support for several contract versions of the same command, which is critical for maintaining compatibility during staged deployment of updates.

3.3 Unified Output Artifact Stream

One of the key reworkings in the transition to the universal architecture was the elimination of output-stream wrappers specific to each pipeline. In the original implementation, each pipeline independently formed the structure of the output message for the Mirera platform, which led to duplication of response-assembly logic: the same field transformations were repeated in the code of different pipelines, while the router became tied to naming details and result formats of specific pipelines. When a new pipeline appeared, the developer had to implement not only the processing logic itself but also duplicate the infrastructure code for assembling the message.

In the new architecture, any service, whether a language model, an optical recognition system, or prospective speech recognition and synthesis services, returns a result as a uniform dictionary with named fields (artifacts). Transformation of

these artifacts into a message for the next step of the chain or into the final response to the platform is performed by a single component, a configurable message transformer that reads transformation rules from the chain description. The transformer supports field renaming, reading and writing artifacts to S3 object storage, simple type conversions, and substitution of default values. This ensures uniform router behavior for all service types and, in particular, makes it possible to use artifacts from both a language-model pipeline and an image-recognition pipeline within a single chain without resorting to special code glue. An additional practical effect is simplified unit testing of pipelines: their correctness is checked by the conformity of the returned artifact dictionary to the schema, without the need to construct the target output-message format for a specific command in each test.

3.4 Pipeline as a Self-Contained Module

Each pipeline is organized as a separate directory containing the minimum necessary set of files: a strict request and response schema based on a declarative description of data structures, an implementation of the execution logic, templates for prompt-forming instructions (for language-model pipelines), or configuration for calls to the service (for pipelines of another modality). The pipeline entry point is an asynchronous function that accepts a validated request, a model instance, and a reference to the service manager; the returned value is a dictionary of named artifacts, which is subsequently processed by the message transformer according to the chain-step description. The use of strong typing for input and output schemas eliminates the possibility of implicit integration errors between steps and ensures rapid detection of inconsistencies already at the request validation stage.

This organization supports hierarchical pipeline lookup, “private - common”: if an implementation specific to the Mirera platform is present in the system, it takes priority; if it is absent, the generalized implementation from the common catalog is used. This makes it possible to reuse common pipelines (for example, a basic audio-recording transcription pipeline) in Mirera-specific chains without code duplication and at the same time to isolate private business logic, such as assignment-formatting features for specific courses, specific limits on response length, and additional postfilters within private pipelines, without cluttering the common catalog with them.

From an organizational point of view, this structure significantly simplifies functional development: adding a new educational task is reduced to creating a new pipeline directory and registering the corresponding command in the

registry of chains and contracts. No changes to the router or service-manager code are required, which reduces the risk of undesirable side effects when releasing new functionality and shortens the time needed to put new educational scenarios into operation.

4. Additional Features of the New Architecture

4.1 Support for Specialized Attachment Types

Unifying the handling of attachments within the ContentProcessor component made it possible to quickly add support for new types of educational materials. In particular, processing of attachments in LaTeX format has been implemented, which is especially important for mathematical, physical, and engineering courses, where task statements, student solutions, and teachers' methodological materials are often written in this markup language. Since a LaTeX file is a structured text document with a fixed encoding, its processing does not require a call to a recognition subservice: the content is decoded immediately and passed directly to the pipeline. A similar fast path is also provided for simple text attachments with extensions such as .txt and others.

The interface for working with all attachment types remains unified: the pipeline receives an already prepared textual representation of the document regardless of whether the original file was processed by a remote recognition subservice, a local fallback parser, or direct decoding. This simplifies the development of pipelines capable of accepting materials in arbitrary formats and relieves the pipeline developer of the need to account for the specifics of each format separately. Adding support for a new attachment type is reduced to describing the method for extracting text and registering a new branch in the extension-handler map within ContentProcessor. This simplifies system development as new types of educational materials appear, for example source code in archives for various programming languages, interactive-task forms, or attachments with specialized markup.

4.2 Error Classification and Handling

In the original architecture, each pipeline was independently responsible for intercepting and interpreting inference errors, which led to code duplication and nonuniform response semantics: errors of the same nature could be classified differently in different pipelines, complicating observability and system support. The new architecture adopts the principle of "pipelines without exception interception": exceptions arising during execution are propagated to the router, which

classifies them through a special error-handler component and decides how the system should respond.

Errors are divided into two main categories. Transient errors (timeout when accessing a model, loss of connection to a model, temporary unavailability of a subservice, or absence of a free model instance in the pool) cause the message to be returned to the queue for repeated processing, subject to a limit on the total number of attempts and an increasing delay between them. Permanent errors (violation of the request schema, absence of the required pipeline, fatal inference error, or incorrect chain configuration) move the request into a failed-processing state: the router forms an error response according to the command contract and enriches it with a code from the catalog of permanent errors. The catalog of permanent errors is maintained as part of the configuration and describes each error category with its code, a short description for logging purposes, and a user-facing message displayed in the Mirera platform. This classification is unified for all service types: the language-model service, the recognition subservice, and prospective speech recognition and synthesis services use the same categorization mechanism, which ensures a uniform system response to error situations regardless of the modality of the processed data and substantially simplifies incident diagnostics for the operations team.

4.3 Unified Metrics and Observability

Implementing the metrics collection method directly in the BaseServiceManager base class ensures uniform collection of telemetry data for any connected service. Metrics are exported to the Prometheus monitoring system [9] through a standard publication endpoint and include counters of received and completed requests broken down by pipeline, processing-time histograms with configurable bucket boundaries for granular latency analysis, indicators of the current state of the model pool (the number of busy and free instances of each model), and error counters categorized by type. On the service side, these metrics are updated automatically as part of standard service-manager operations and do not require any additional code from the developer of a specific derived class.

Because the set of metrics and their semantics are identical for all services, monitoring and alerting are configured once and then apply to any future services. This relieves the developer of a new service manager from the task of designing custom telemetry and significantly simplifies the work of the operations team: uniform Grafana dashboards are used for all services, and alert thresholds are formulated in terms independent of the model type. An additional practical effect is the ability to trace request processing end to end across the entire

sequence of steps, including steps that cross service boundaries: unified request identifiers and agreed semantics of timing metrics make it possible to build complete time diagrams of processing and quickly localize bottlenecks in composite processing chains.

4.4 Architecture Readiness for Connecting Speech Services

As part of the reworking, frameworks have been laid for future speech recognition and speech synthesis services. The need for them is driven by the development of lecture and seminar learning formats in the Mirera platform, as well as by parallel work on the digital teacher twin [4], which already uses speech recognition and synthesis components as part of an independent system. The service-factory registry provides the corresponding connection points for automatic speech recognition (ASR) and text-to-speech synthesis (TTS) services. The processing-chain configuration already describes examples of complex multimodal chains, in particular a “class audio recording transcription - summarization of the resulting text by a language model” chain and a more complex “transcription - summarization - speech synthesis based on the summarized text” chain, which combines three services of different modalities in a single processing sequence. The corresponding rules for field transformation between steps and for working with object storage for large audio and text artifacts have already been formulated, agreed upon, and tested on mock implementations.

At the current stage, the speech recognition service and the speech synthesis service are

implemented as interface stubs that correctly fulfill the BaseServiceManager contract for the purposes of integration testing and checking chain integrity, but return an unimplemented marker when actual request processing is attempted. After the stubs are replaced with full implementations of the corresponding models, connecting the new service type will occur without changes to the router code or to service managers of other modalities. This is the main practical result of the architectural evolution carried out and confirms the viability of the proposed approach.

5. Conclusion

This paper describes the evolution of the neural network module architecture of the Mirera platform, providing universal support for multimodal AI services. It is shown that the original LLM-oriented core had fundamental limitations.

The proposed architecture eliminates these limitations through a set of interrelated solutions: an abstraction with a unified contract for services of any modality, moving processing chains and the platform-interaction contract into configuration files, and unifying the output artifact stream with its transformation through a configurable transformer.

In the future, it is planned to expand the supported modalities and the set of scenarios involving new models, as well as to improve and optimize the current architecture.

The work was carried out within the framework of the state assignment of the National Research Center “Kurchatov Institute” - SRISA, topic No. FNEF-2024-0001 (1023032100070-3-1.2.1).

References

1. Chu Z., Wang S., Xie J., Zhu T., Yan Y., Ye J., Zhong A., Hu X., Liang J., Yu P. S., Wen Q. LLM agents for education: Advances and applications // arXiv preprint arXiv:2503.11733. - 2025. URL: <https://arxiv.org/pdf/2503.11733> (accessed: 02.07.2026).
2. Mirera Platform [Electronic resource]. - URL: <https://www.mirera.ru> (accessed: 02.07.2026).
3. Leonov A. G., Martynov N. S., Mashchenko K. A., Paremuzov M. S., Pchelin K. K., Shlyakhov A. V. Areas of application of large language models for digital educational platforms // Proceedings of SRISA. - 2025. - Vol. 15, No. 2. - P. 9-15. URL: <https://www.t-niisi.ru/jour/article/view/72/120> (accessed: 02.07.2026).
4. Kushnirenko A. G., Leonov A. G., Mashchenko K. A., Martynov N. S., Pchelin K. K., Shlyakhov A. V. Integration of a neural network assistant for programming learners into a digital educational platform // Informatics and Education. - 2026. - Vol. 41, No. 1. - P. 23-36. URL: <https://info.infojournal.ru/jour/article/view/1426/783> (accessed: 02.07.2026).
5. Leonov A. G., Mashchenko K. A., Martynov N. S., Pchelin K. K., Shlyakhov A. V. Automated generation of educational video materials based on a modular digital teacher twin system // Software & Systems. - 2025. - Vol. 38, No. 4. - P. 694-703. URL: <https://swsys.ru/files/2025-4/694-703.pdf> (accessed: 02.07.2026).
6. Gridnev N. V., Karavaeva A. S., Leonov A. G., Mashchenko K. A., Pchelin K. K., Tarasuk E. D.

Hybrid learning model: innovations in attendance tracking in the Mirera digital educational platform // Proceedings of SRISA. - 2025. - Vol. 15, No. 1. - P. 58-64. URL: <https://www.t-niisi.ru/jour/article/view/10/20> (accessed: 02.07.2026).

7. Vinoski S. Advanced Message Queuing Protocol // IEEE Internet Computing. - 2006. - Vol. 10, No. 6. - P. 87-89. - DOI: 10.1109/MIC.2006.116.

8. Auer C., Lysak M., Nassar A., Dolfi M., Livathinos N., Vagenas P., Berrospi C., Omenetti M., Lindlbauer F., Dinkla K., Mishra L., Kim Y., Gupta S., de Lima R. T., Weber V., Morin L., Meijer I., Kuropiatnyk V., Staar P. W. Docling Technical Report // arXiv preprint arXiv:2408.09869. - 2024. URL: <https://arxiv.org/pdf/2408.09869> (accessed: 02.07.2026).

9. Brazil B. Prometheus: Up & Running: Infrastructure and Application Performance Monitoring. - 2nd ed. - O'Reilly Media, 2024. - 506 p.

Поступила в редакцию / Received: 25.06.2026.

Поступила после рецензирования / Revised: 30.06.2026.

Принята к печати / Accepted: 03.07.2026.