# Development of a Web Application for Automating the Crane Operator's Workstation in an Electric Steelmaking Shop with a Crane and Ladle Movement Visualization System

Rauza R. Abdulveleeva
*Mathematics and Science dept.*
*Novotroitsky branch of the federal state autonomous educational institution of higher education "National Research Technological University MISIS"*
Novotroisk, Russia
rashitovna-2011@mail.ru

Vladyslav A. Kravchenko
*Novotroitsky branch of the federal state autonomous educational institution of higher education "National Research Technological University MISIS"*
Novotroisk, Russia
kravasos18@gmail.com

Ildar R. Abdulveleev
*Power supply of industrial enterprpises dept.*
*Nosov Mangnitogorsk State Technical University*
Magnitogorsk, Russia
i.abdulveleev@magtu.ru

2025 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) | 979-8-3315-1219-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICIEAM65163.2025.11028389

***Abstract*—The article presents a web application for automating the crane operator's workstation in an electric steelmaking shop, integrating management tools and visualization of crane and ladle movements into a unified environment. The relevance of the development is due to the insufficient transparency of crane operations, which leads to equipment downtime, disruptions in the technological process, and a decrease in overall production efficiency. The proposed solution is implemented using a combination of Django and FastAPI frameworks for the backend, with an SQL database (PostgreSQL) and NoSQL storage (Redis). The frontend part of the application is developed using jQuery and Bootstrap, while Celery is used for background tasks. System testing is performed using pytest. The implementation of the automated workstation increases work speed, improves response to production incidents, and provides flexibility in crane control. The article describes the application architecture, project file structure, and data flow diagrams. The implementation of the system will increase production by reducing downtime and decreasing the number of violations caused by insufficient transparency in crane operations.**

***Keywords—automated crane operator's station, web-application, mimic diagrams***

## I. INTRODUCTION

Automation plays a key role in modern industrial production, providing tools for more efficient management and optimization of production processes. Many enterprises face challenges related to insufficient automation, which leads to increased time and financial costs, as well as reduced productivity and product quality [1]. In this context, automation becomes a relevant topic, as it enhances flexibility, improves staff responsiveness, and reduces production costs [2-4].

One of the main problems in electric steelmaking shops is the lack of transparency in crane operations, which leads to equipment downtime and disruptions in the technological process. According to the "ESPC Report for Six Months" from July 2022 to November 2022, an average of 1204 minutes of waiting time for smelting was recorded in the casting bay of the electric steelmaking shop, resulting in process violations and over 18 hours of downtime due to the absence of ladles and crane occupancy [5]. Similar problems are common in many metallurgical enterprises, where the lack of monitoring and visualization systems for crane and ladle movements leads to significant losses of time and resources [6].

The implementation of automated control and monitoring systems, such as MES systems, has already proven its effectiveness at large metallurgical enterprises. For example, PJSC "Magnitogorsk Iron and Steel Works" (MMK) implemented MES systems using RFID tags, which reduced metal losses by 1%, increased productivity by 5% (600,000 tons of steel per year), and reduced electricity costs by 5% [7]. Such results demonstrate that the automation of crane and ladle management processes can significantly improve production performance [8].

The development of a web application for automating the crane operator's workstation with a crane and ladle movement visualization system is a relevant task because it allows:

- Increasing the transparency of crane and ladle operations;
- Reducing equipment downtime;
- Improving response time to production incidents;
- Ensuring real-time control over the technological process [9].

Thus, the implementation of such a system will not only increase the efficiency of the electric steelmaking shop but also provide the enterprise with a competitive advantage in the metallurgical market [10].

979-8-3315-1219-4/25/$31.00 ©2025 IEEE

## II. Description of the Application

To control crane movement, the decision was made to use RFID tags. This technology was chosen due to its resilience to harsh workshop conditions, such as vibration, high dust levels, and elevated temperatures [9]. RFID technology has proven its effectiveness in industrial environments, providing accurate real-time tracking of objects [6].

The web application is presented in the form of a mimic diagram, created using a combination of Python frameworks, Django and FastAPI. Django, with its MVT (Model-View-Template) architecture, allows for efficient separation of the application's logic into models, views, and templates, simplifying development and maintenance [11]. FastAPI, in turn, provides high performance and ease of creating APIs for integration with other systems [12].

A brief file structure of the web application with a detailed description of the automated workstation application is shown in Figure 1.

The diagram in Figure 1 is divided into logical groups: red color marks the root packages containing applications and the entire project. Purple color ─ structures related to testing. Yellow color ─ project resources: frameworks, libraries. Blue color ─ modules. Green color ─ packages of the application for implementing the automated workstation for the crane operator.

The "uralsteel" package ─ the root of the entire project, highlighted in red in Figure 1. It includes some auxiliary packages, such as the "tests" package, containing the part related to testing, highlighted in purple, "venv", including the virtual environment of the web application ─ all necessary frameworks and libraries, highlighted in yellow. There are also two modules ".env" and ".test.env", which hide credentials, tokens, and data necessary for the application's operation, for the dev and test environments, respectively.

The "src" package (highlighted in red in Figure 1) contains the actual code. It includes the "api" package with the API service.

The "uralsteel" package ─ the focus of this article, is the part with Django applications. All sub-packages are marked in green. Let's consider the packages inside. All packages in Figure 1 are highlighted in blue:

- "media" ─ stores all media files.;

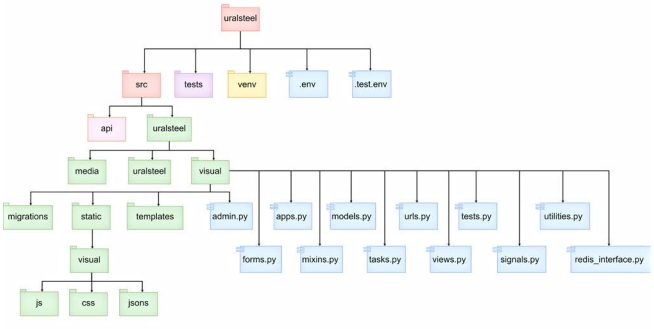


Fig. 1. File structure of the mimic diagram.

- "uralsteel" ─ package with the configuration of the entire Django project, containing routes of all pages in a single "router", Celery settings, settings for asynchronous and regular servers.

- "visual" ─ is a package containing the application with tools for automating the crane operator's workstation.

The "migrations" package is related to database migrations, the "static" package contains js and css code, the "templates" package includes HTML page templates for the application, as well as the following modules:

- "admin.py" ─ configuration and setup of the admin panel.

- "apps.py" ─ class launching the visual application.

- "models.py" ─ application models, its business logic, and entities.

- "urls.py" ─ page routes of the application (configuration will be accumulated in the main "router" of the web application).

- "tests.py" ─ application tests.

- "utilities.py" ─ module with common utilities of the application.

- "forms.py" ─ module with configuration of all forms of the application. They will be displayed on HTML pages.

- "mixins.py" ─ module with mixin classes (metaprogramming). Such classes will "mix in" the necessary functionality into the application views, for example, CSRF token functionality for page protection.

- "tasks.py" ─ module with background tasks.

- "views.py" ─ application views, the controller layer.

- "signals.py" ─ special module that "listens" to web application signals and triggers certain logic upon certain signals, for example, when a unit breaks down.

- "redis_interface.py" ─ common interface for working with Redis storage.

The application architecture is based on the classic MVT (Model-View-Template) model, which is widely used in modern web applications [13]. The Model layer is responsible for business logic and database interaction, the View layer processes requests and passes data to templates, and the Template layer handles data visualization [14].

For caching data and handling background tasks, Redis and Celery are used. Redis has proven its efficiency as a cache storage and message broker, which helps reduce the load on the main database [9]. Celery, in turn, ensures the execution of asynchronous tasks, such as report generation and notification distribution [15].

The architecture diagram is shown in Figure 2.

The Model Layer serves as the interface for the domain model, where the domain is defined as a collection of objects related to the application's business logic. In Django, models

implemented in the classes of the models.py module (as illustrated in Figure 2 with a corresponding label) encapsulate the rules of business logic and data handling methods. This layer describes the data structures that the application interacts with, including cranes, ladles, electric steelmaking shop units, users, and user groups. As an example of the application's domain model, Figure 3 presents a fragment of the source code containing the ladle model.

Each business logic object has such a descriptive model. This is the Django implementation of the ORM (Object-Relational Mapping) approach. Here, the application model is essentially linked to the database table. In the class attributes, column settings, column constraints are configured, and the standard method "str" is necessary for logging, responsible for how the class object will be displayed.

One of the interesting parts is the nested Meta class. This class is used for metaprogramming in Django. Here, more flexible model configuration occurs, for example, the "verbose_name" attribute is responsible for a user-friendly class name. In this class, you can also configure collection sorting, abstraction type, and many other aspects. For example, Figure 4 shows the configuration of the abstract JWT token model — TokenBase, whose functionality is inherited by two models: RefreshToken (token for obtaining a new access token) and TokenBlackList (model for tokens in the blacklist, needed to provide additional protection in case of theft of refresh and access tokens).

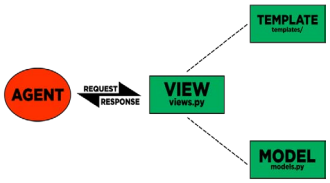


Fig. 2. Architecture diagram of the workstation application.

```
class Ladles(models.Model):  5 usages

    title = models.CharField(verbose_name="Ladle name", max_length=100)
    is_active = models.BooleanField(default=False, null=False, blank=False,
                                    verbose_name="Is ladle acting")
    is_broken = models.BooleanField(default=False, null=False, blank=False,
                                    verbose_name="Is ladle broken")

    def __str__(self):
        return f'{self.title}'

    class Meta:
        verbose_name = 'Ladle'
        verbose_name_plural = 'Ladles'
```

Fig. 3. Domain model of a ladle.

```
class ArchiveDynamicTable(DynamicTableAbstract):  6 usages

    def __str__(self):...

    class Meta:
        verbose_name = 'Archive melt'
        verbose_name_plural = 'Archive melts'
        ordering = ('-actual_end',)
```

Fig. 4. Configuration of the abstract JWT token model.

View layer (presentation layer) - this is the infrastructure layer. This layer complements the trivial tasks of the controller layer, namely, it additionally takes on the responsibility of transferring information to the template layer, that is, to the template. In this context, requests are processed, data is prepared, data flows are managed, and so-called "use cases" — system usage scenarios, that is, the business logic of the application, are called.

Figure 5 shows the configuration of sorting the smelting archive by the date of smelting completion.

View classes are stored in the "views.py" file. Let's consider a specific example of a View class implementation in the web application. A fragment of the program code - the View class of the crane operator's automated workstation page is shown in Figure 6.

Python's object-oriented design provides another important metaprogramming tool — "Mixin" classes. In LadlesView, these are the RedisCacheMixin and TemplateView classes, which are not directly inherited but rather "mix in" useful functionality into our view. RedisCacheMixin — a developed interface of the entire mimic diagram project, contains gateways for working with Redis cache storage, all CRUD operations. The RedisCacheMixin interface is shown in Figure 7.

```
class TokenBase(models.Model):  2 usages
    refresh_token = models.CharField(max_length=1000,
                                     unique=True,
                                     verbose_name='Token')
    expire_date = models.DateTimeField(verbose_name='Expiration date')
    token_family = models.UUIDField(
        default=uuid.uuid4,
        editable=False,
        verbose_name='Unique identifier of token family',
    )

    def __repr__(self):...

    class Meta:
        abstract = True


class RefreshToken(TokenBase):...


class TokenBlacklist(TokenBase):...
```

Fig. 5. Configuration of the archive model sorting.

```
class LadlesView(LoginRequiredMixin, RedisCacheMixin, TemplateView):  15 usages
    template_name = 'visual/ladles.html'

    def get(self, request, *args, **kwargs):...

    def post(self, request, *args, **kwargs):...

    @staticmethod  2 usages
    def time_convert(time: str) -> datetime:...

    @staticmethod  1 usage
    def get_ladles_info(date: datetime) -> dict:...

    @staticmethod  3 usages
    def ladles_into_dict( ladles_queryset, ladles_info: dict, is_transporting: bool = False, is_plan: bool = False) -> dict:...

    @staticmethod  2 usages
    def from_active_to_archive(operation: Type[ActiveDynamicTable]) -> None:...
```

Fig. 6. View class of the crane operator's automated workstation page.

The project uses redis-stack version with modules for working with JSON files. That is, the crane operator's

workstation caches typical responses and reduces the load on the database and server.

In LadlesView LoginRequiredMixin is needed to implement authorization processes on the page, and TemplateView is needed to link the template with URL addresses, render the context and template. The class attributes configure the template for the crane operator's workstation page, and the get and post methods exist to respond to GET and POST requests. Other static methods are used in the get and post methods.

Template layer - this is the layer of presenting prepared information. The difference between view and template is quite tangible. Views represent what information we use, and the template layer ─ how exactly and in what form we present it to the agent. The types of contextual information presentation are collected in the templates package, as shown in Figures 1 and 2. Let's consider an object of the template layer ─ the template of the crane operator's workstation page (Figure 8).

The template inherits from the base template containing common parts for all pages, for example, header and footer. Also, the template actively uses directives (blocks marked with "{{" and "}}" symbols on both sides) and Django template tags (blocks marked with "{%" and "%}" symbols on both sides) to reduce code and comply with DRY and KISS principles.

```
class RedisCacheMixin:  4 usages
    @staticmethod  3 usages
    def get_key_redis_json(key_name: str) -> dict | None:...

    @staticmethod  3 usages
    def set_key_redis_json(key_name: str, data: dict, ttl: int) -> None:...

    @staticmethod  1 usage
    def get_key_redis(key_name: str) -> str | None:...

    @staticmethod  1 usage
    def set_key_redis(key_name: str, data: str, ttl: int) -> None:...

    @staticmethod
    def delete_key_redis(key_name: str) -> None:...

    @staticmethod  1 usage
    def delete_keys_redis(pattern: str) -> None:...
```

Fig. 7. RedisCacheMixin.

```
{% extends 'layouts/schema.html' %}

{% load static %}

{% block addlinks %}
<script src="{% static 'visual/js/ladles.js' %}"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/leader-line/1.0.7/leader-line.min.js" integrity="sha512-0dNdzMjpT6p
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.min.js" integrity="sha384-B8tl+e6JRgqQAUMxJ
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" integrity="sha384-
<script...>
{% endblock addlinks %}

{% block inputtime %}
<div class="center-container"...>
{% endblock inputtime %}

{% block modal %}
<div class="modal fade" id="ladlesModal" tabindex="-1" aria-labelledby="ladlesModalLabel" aria-hidden="true"...>

<button type="button" class="btn btn-primary" id="modal-button" data-bs-toggle="modal" data-bs-target="#ladlesModal" hid

<div class="modal fade" id="ladlesDispatcherModal" tabindex="-1" aria-labelledby="ladlesDispatcherModalLabel" aria-hidde

<button type="button" class="btn btn-primary" id="modal-dispatcher-button" data-bs-toggle="modal" data-bs-target="#ladla
{% endblock modal %}
```

Fig. 8. Template of the crane operator's automated workstation page.

Let's consider the way the agent and the web application interact using the example of the crane operator's workstation page. The agent requests information via the URL route of the workstation page, Django processes this request, identifies it by the route and HTTP method, after which intermediaries are executed, for example, one of the intermediaries processing the CSRF token. After executing the intermediaries, control passes to the view layer configured for this route, in this case ─ LadlesView. LadlesView interacts with the domain using the model layer, in the case of the workstation ─ the Ladles model. The penultimate step is to transfer the context to the template layer (Figure 9) to form the templates. And the last step ─ executing the intermediaries and sending the response with the HTML page to the agent. The DFD diagram of the data flow inside the crane operator's automated workstation application is shown in Figure 9.

## III. Operation of the Crane Operator's Automated Workstation

The process of authentication and authorization in the application is implemented using JWT tokens, which ensures security and ease of integration with other systems [16]. Real-time visualization of crane and bucket movements allows for prompt response to changes in the production process, reducing downtime and increasing overall efficiency [17, 18].

The process of starting work with the web application begins with the mandatory procedure of authentication and authorization, only then access to the application's functionality will appear. If it is an integration using the API, then it is necessary to obtain a JWT authorization token through a separate API. The authentication process is shown in Figure 10.

After authentication, the authorization process will be automatically performed, if the agent has the necessary rights, they will gain access to the main functionality.The crane operator's automated workstation page is shown in Figure 11.

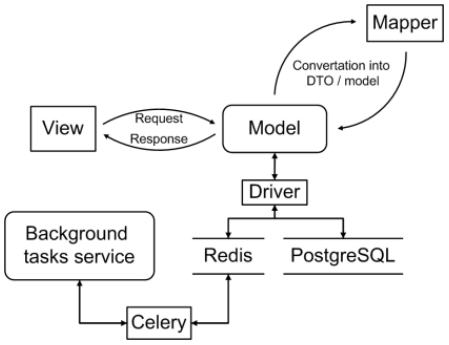


Fig. 9. DFD diagram of the data flow inside the crane operator's automated workstation application.

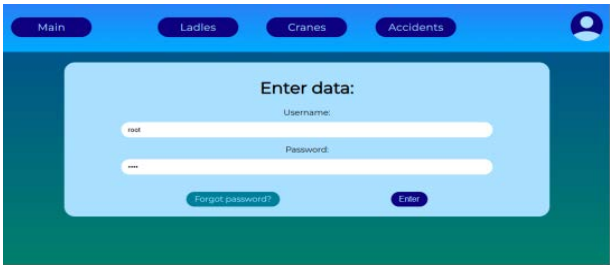


Fig. 10. Agent authentication.

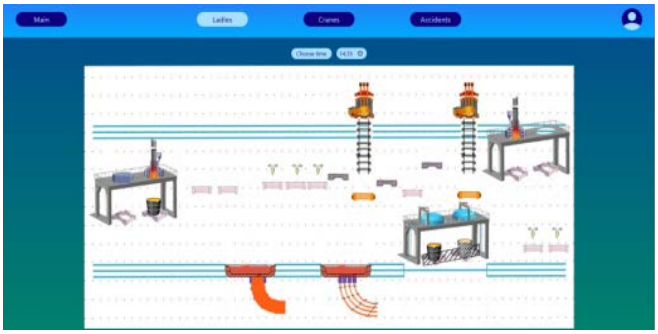

Fig. 11. Crane operator's automated workstation page.

On the mimic diagram, you can see ladles of several types. There are three such types in total. The first type ─ "starting". This is a ladle starting another smelting iteration, it is shaded on the mimic diagram. This ladle is waiting for confirmation of the start of the operation, after confirmation it will automatically be brought to the second type, The second type ─ "waiting" ─ this is a ladle waiting for the completion of the operation. On the mimic diagram, it is displayed as a regular ladle. After confirming the completion of the operation, it will be brought to the third type. The third type ─ "transportable" ladle. This is a ladle that needs to be transported to the next unit. It is displayed as a ladle with an arrow leading to the next unit. Confirmation to activate a ladle of any type can be made by right-clicking on the ladle icon in a pop-up modal window, thus confirming by clicking that the modal window is created to prevent accidental clicks. For each type of ladle, its own message is prepared, all types of confirmation modal windows are shown in Figure 12.

To view the characteristics of each ladle, you can left-click on its icon. The characteristics of one of the ladles are shown in Figure 13.

Also, a time selection form has been added to the mimic diagram page. It was created within the framework of modulating the smelting process to simulate the passage of time, as production processes can be very long and testing such systems in real-time is not rational. Let's consider a situation where one ladle has completed an operation on the current unit. Let it be a ladle standing on the steel vacuuming unit at the first position. First, it is necessary to move the time, let's simulate a situation where the ladle is delayed by 5 minutes, respectively, we will move the time to "14:50". After confirming the completion of the operation according to the previously mentioned algorithm, the ladle will become "transportable", the crane operator will receive a task to transport it. The "transportable" ladle is shown in Figure 14, an arrow with a description of the next position comes from it, in this case, it is MMLZ.

After the completion of the smelting process, the information will be entered into the archive.

Thus, the algorithm of the crane operator's automated workstation in the form of an event-driven process chain (EPC) model is shown in Figure 15.

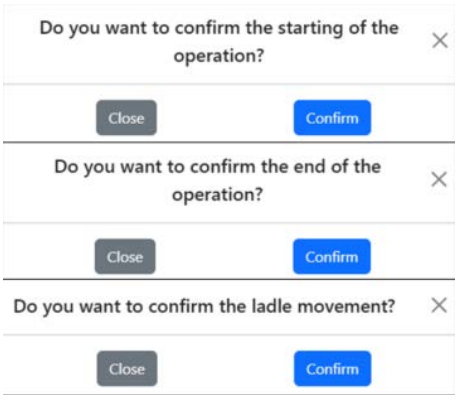


Fig. 12. All types of confirmation modal windows.


Fig. 13. Ladle characteristics.

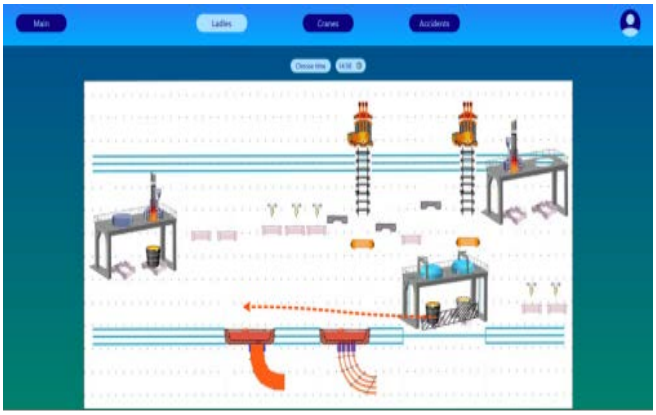

Fig. 14. Transportable ladle.

## CONCLUSIONS

The article proposes and describes a web application for automating the crane operator's workstation in an electric steelmaking shop, integrating management tools and visualization of crane and ladle movements into a unified environment. The development of such a solution is relevant in the context of modern metallurgical production, where insufficient transparency in crane operations leads to equipment downtime, disruptions in the technological process, and a decrease in overall efficiency.

The use of modern technologies, such as Django and FastAPI for backend development, PostgreSQL and Redis for data storage, and Celery for background tasks, has enabled the creation of a flexible and scalable solution. The implementation of a visualization system based on RFID tags

ensures accurate real-time tracking of crane and ladle positions, significantly improving the transparency and controllability of the production process.

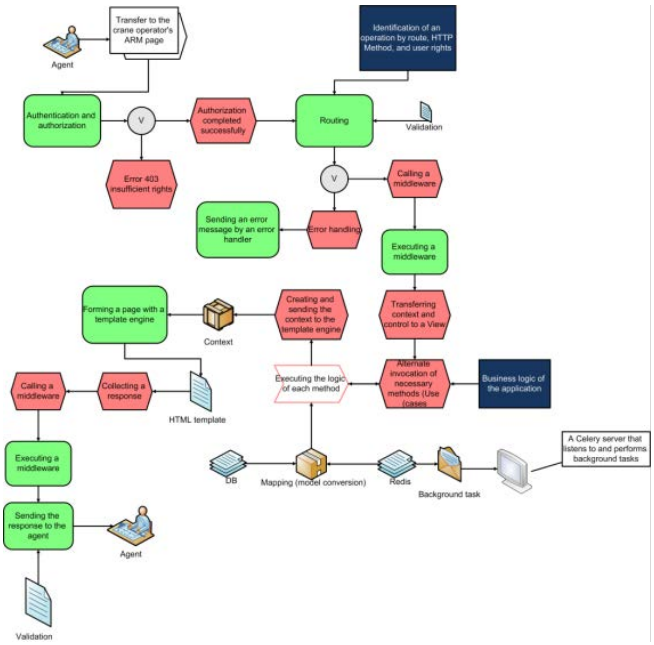


Fig.15 EPC diagram of the crane operator's automated workstation algorithm.

The proposed solution enhances the transparency of crane and bucket operations, reduces downtime, and increases productivity. The expected economic impact from implementing the system amounts to 16 million rubles per year, achieved by stabilizing the speed of the continuous casting machine (CCM) and reducing downtime by 45 hours annually [8].

Thus, the proposed web application not only addresses the key problems of the electric steelmaking shop but also provides significant economic benefits. Further development of the system may include integration with other production systems, such as MES and ERP, as well as expanding functionality to automate other production areas.

Overall, the development and implementation of an automated crane operator's workstation demonstrate that modern IT solutions can significantly improve the efficiency of industrial processes, reduce costs, and provide a competitive advantage for enterprises in the market.

## REFERENCES


[1] M. P. Groover, Automation, Production Systems, and Computer-Integrated Manufacturing. Pearson, 2019.

[2] L. Wang, M. Törngren, and M. Onori, “Current status and advancement of cyber-physical systems in manufacturing,” Journal of Manufacturing Systems, no. 37, pp. 517–527, 2015.

[3] R. R. Abdulveleeva, V. A. Korsakov, and I. R. Abdulveleev, “Crane movement modulator for web application of electric steel melting shop mnemonic scheme with crane movement visualisation system,” 2024 International Ural Conference on Electrical Power Engineering (UralCon), Magnitogorsk, Russian Federation, 2024, pp. 812–816, doi: 10.1109/UralCon62137.2024.10718976.

[4] R. R. Abdulveleyeva and V. Korsakov, “Comparative analysis of methods and means of visualisation of cranes movement of electric steelmaking shop of JSC ‘Ural Steel’ for use in web application,” Digital Systems and Models: Theory and Practice of Design, Development and Application. Kazan, April 2024, pp. 270–274.

[5] Report “ESPTS Six-Month Report”, 2022. PJSC "Magnitogorsk Iron and Steel Works".

[6] Y. Zhang and G. Zhang, “Real-time visualization and monitoring of industrial processes using web-based technologies,” Computers in Industry, no. 91, pp. 1–12, 2017.

[7] LLC "MMK-Informservice", “Implementation of MES systems at PJSC 'MMK'.” Magnitogorsk, Russia, 2021.

[8] X. Li and L. Gao, “An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem,” International Journal of Production Economics, no. 174, pp. 93–110, 2016.

[9] R. Want, “An introduction to RFID technology,” IEEE Pervasive Computing, no. 5(1), pp. 25–33, 2006.

[10] S. Nahmias and Y. Cheng, Production and Operations Analysis. McGraw-Hill, 2009.

[11] W. Vincent, Django for Professionals. San Francisco, CA: Django for Beginners LLC, 2021.

[12] M. Grinberg, Flask Web Development: Developing Web Applications with Python. O'Reilly Media, 2018.

[13] S. Nahmias and Y. Cheng, Production and Operations Analysis. McGraw-Hill, 2009.

[14] W. McKinney, Python for Data Analysis. O'Reilly Media, 2017.

[15] Celery: Distributed Task Queue, 2023. Official Documentation.

[16] L. Richardson and S. Ruby, RESTful Web APIs. O'Reilly Media, 2013.

[17] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Doctoral dissertation, University of California, Irvine, 2000.

[18] R. R. Abdulveleeva and V. Kravchenko, “Vulnerability analytics of web application mnemonic scheme of electric steel-smelting shop with visualisation system of cranes and steel ladles movement,” Digital Systems and Models: Theory and Practice of Design, Development and Application, Kazan, April 2024, pp. 275–279.