AI Agents in Automation

How beneficial is AI in industrial automation today, and what are the prerequisites?

vorausrobotik

Carl-Buderus-Str. 7 | 30455 Hanover | Germany | vorausrobotik.com

Summary This white paper shows how generative AI, and AI agents in particular, can already deliver real economic benefits in industrial automation beyond the current hype. A pick-and-place application is examined, in which the pick rate (successful gripping of small plastic bags) was increased to over 99 % and the cycle time was reduced by more than 16 %. In addition, significant time savings of up to 80 % in software development and deployment were gained. Furthermore, the current limitations of generative AI as well as the necessary prerequisites for efficient usage in industrial automation are discussed.

1. Introduction and Motivation

In our last white paper "Industrial DevOps", we looked at the importance of modern software development in industrial automation. The consistent programming of all OT (operational technology) components of an automation cell in high-level language (e.g. Python or Rust) and the virtualisation of these components for simulation, testing, and optimisation enable fast software release cycles with high quality. Based thereon, this white paper discusses how AI can be used to optimise industrial automation solutions. Therefore, high test coverage is essential in order to update plant control systems without the risk of accidental damage and to operate them with high availability in the field (industrial grade). We consider different approaches to increase the output of a pick-and-place cell with the help of AI. The goal is not to identify the best possible AI tool – these are subject to rapid changes anyway – but rather on how these modern tools can be integrated into the Industrial DevOps workflow to increase efficiency and what prerequisites must be met for their industrial use.

Traditional programming of industrial automation solutions is reaching its capacity limits, due to high demands of experts with specialised programming skills (structured text, KUKA KRL, FANUC KAREL, ...) and proprietary tool knowledge. This is even more tightened due to an increasing degree of automation and the growing complexity of production facilities and processes. The consequences are high costs and long project durations, as well as pressure on management (primarily CTOs and CDOs) to identify and implement measures to ensure competitiveness.

In this context a promising solution is generative AI embedded in the Industrial DevOps workflow. This may include tools to assist with troubleshooting on the shop floor, requirements management, or automatic documentation. AI agents that optimise existing solutions (i.e. increase the efficiency of already installed systems) or automatically generate software, thereby drastically reducing development time, offer particular economic potential. The latter two approaches have in common that the code generation is not always transparent to the user and that this code controls processes on expensive hardware. Errors or gaps in the software can therefore lead to damage and downtime, contradicting the efficiency gains. Here, high test coverage is of particular importance, although this is difficult to achieve due to numerous corner- and edge-cases.

In this white paper, we look at industrial relevant examples of efficiency improvements. In addition, the necessary framework for the use of automatically optimised or generated code in automation cells is analysed. Generative AI tools (so-called large language models) are the preferred software tools for this purpose. As they are an integral part of AI agents, we will first take a look at how they work.

2. Generative AI for Industrial Automation

This section briefly outlines the basic functioning of large language models (LLMs) and the associated challenges.

2.1 How Generative AI Works

Typically, LLMs are a special form of deep neural networks. The input (prompt) is broken down into sub-fragments (tokens) and, based on this, a probability-based prediction for the output is calculated. This can be natural language, but formulas or software are also possible. Large amounts of data are required to train the neural network and thus to optimize the probabilities. To this end, mainly publicly available material on the internet is therefore exploited.

A special form are so-called RAG (retrieval augmented generation) models. Here, the LLM is enriched with proprietary (local) information available in the company in order to provide more specific answers. This also reduces the probability of hallucinations (see section 2.2).

2.2 The Particular Challenges

Security and Reliability

The use of material available on the internet to train LLMs is also one of their greatest weaknesses: the quality of the LLM's output is heavily dependent on the quality and quantity of the training data. If only limited data are available or if they are of poor quality (e.g. poor programming style, errors in formulas), the LLM learns from this information and may produce poor/unusable results. Furthermore, LLMs do not understand the

temporal behaviour of physical components that is important for OT applications. The well-known phenomenon of hallucinations, i.e. the generation of content that is not part of the training data but nevertheless appears to be correct, is also a current problem. If code generated in this way is deployed on the real system without extensive testing, damage and downtime are inevitable. This severely limits industrial use – an inaccurate dashboard may still seem (somehow) acceptable, but a faulty machine control system definitely does not.

There is also a risk that the code contains security flaws (which may already be known or fixed), making the system vulnerable to cybercrime. The reason for this is that training is very costly and time-consuming and only takes information up to a certain cut-off date into account – meaning that latest information may not be included. This poses a major challenge, particularly in the context of the CRA (Cyber Resilience Act). In theory, it would even be conceivable for malicious code to be deliberately introduced.

Integration into Existing IT/OT Landscapes

Typically, industrial applications do not run centralized on a single computer, but are distributed across several control components: e.g. programmable logic controllers (PLCs) coordinate conveyor belts, a robot controller commands robot movements, and a head controller orchestrates the cell and communicates with a smart camera. The challenge lies in the proprietary programming languages involved (e.g. structured text and robot-specific languages) and the individual computer and communication topology. Here, the automatically generated code must fit in as a customized solution, taking into account further additional manufacturer-specific features (e.g. Siemens vs. Rockwell vs. ...). In addition, as mentioned above, sufficient training material is required. From an AI perspective, a centralised and uniform control of all components (requiring less contextual knowledge) in high-level languages (for which there is correspondingly more training data available) would be the more promising choice.

Legal Aspects and Intellectual Property

According to current (German) legal opinion, the result generated by AI is not subject to copyright law, as this requires a personal creation. However, if, for example, code that is present in the training data is simply replicated, this may constitute a copyright infringement if the corresponding training data themselves are protected by copyright (source: https://kpmg-law.de/ki-und-urheberrecht-was-ist-beim-einsatz-von-llms-erlaubt/). RAG architectures that draw on an internal knowledge base have the potential to reduce the risk of copyright infringement, even if it cannot be completely avoided.

While the copyright aspects are difficult to grasp and, in some cases, still need to be clarified in court of last resort, liability issues in the event of malfunctions are much more clearly regulated: the revision of the Product Liability Directive in 2024 significantly extends the liability of software manufacturers. A violation of the Cyber Resilience Act (CRA) can also give rise to liability (source: https://www.heise.de/hintergrund/Software-Anbieter-aufgepasst-Sie-haften-jetzt-fuer-fehlerhafte-Produkte-10027145.html). In this

respect, automated software testing plays an essential role. Manual testing is too time-consuming and cost-intensive and, therefore, does not allow for rapid update cycles as requested.

3. Simulation and High-Level Language as Enabler

What can be deduced from the challenges mentioned in section 2?

It is essential that a large amount of data is available for training the LLMs, as probabilities are optimised. This condition is easier to fulfil if the automation is programmed in common high-level languages, which are much more widely available on the internet than, for example, structured text, KRL, etc. This also helps to balance the problem of faulty code examples in LLM training mentioned in section 2.2 – due to the large amount of training data in high-level language, faulty code snippets are statistically much less significant and the quality of the networks is thus increased. Similarly, a central, real-time-capable control computer for the entire cell simplifies the code generation process, as the individual (computer/software/communication) architecture of the cell does not have to be taken into account. A suitable solution for this is the **voraus.core** control platform (see figure 1). A modern software architecture allows all components of an automation cell to be addressed via standardised interfaces (EtherCAT, PROFINET, OPC UA, etc.) in high-level language and in real time.

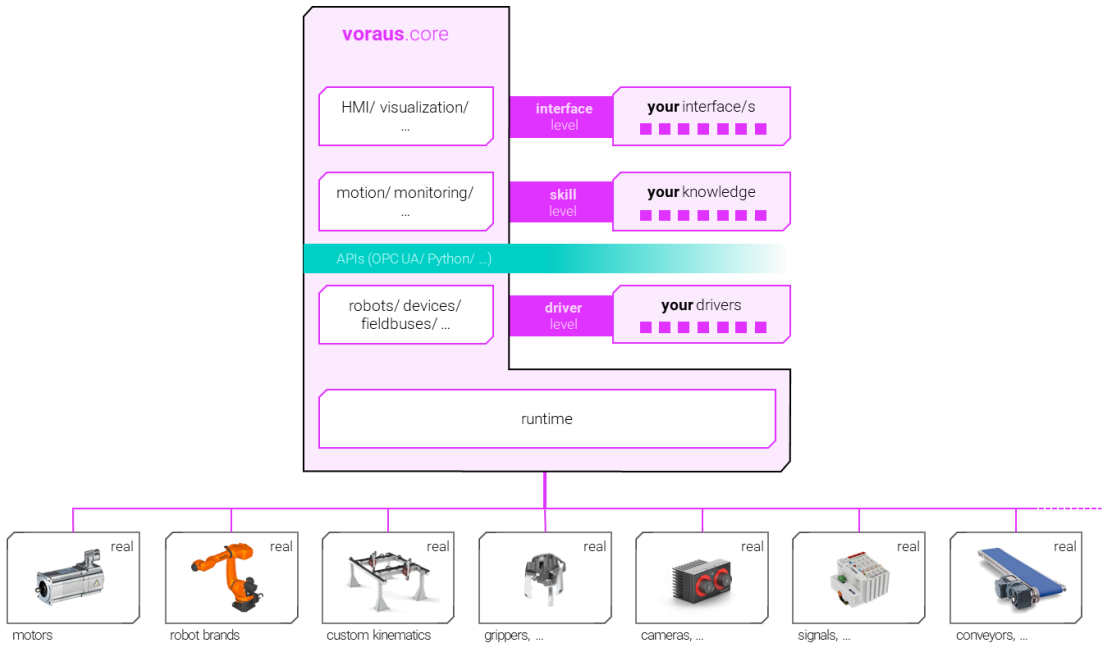

Figure 1: voraus.core control platform: modern architecture and open interfaces enable direct communication with fieldbus components via high-level languages and in real time.

However, the question of quality and completeness of the optimised or generated code remains still open. The numerous edge- and error-cases play a particularly important role, as they impede high test coverage, but are essential for avoiding downtime or even machine damage (and the associated liability risk). It is therefore undisputed that the generated code must be tested extensively before it is deployed on the real system. A simulation environment that provides the individual automation components in virtualised form and is programmed in the same language as the real cell is the solution: This is the only way to test the automation application in its entirety (and not just in fragments, as described above) and achieve high test coverage at an early stage of the development process. It is important that the same code base is used as in the real cell. Translation into manufacturer-specific programming languages prior to deployment drastically reduces the significance of the test results. The **voraus.pioneer** development environment (see figure 2) implements this approach and ensures a uniform code base between development, simulation, and real cell.

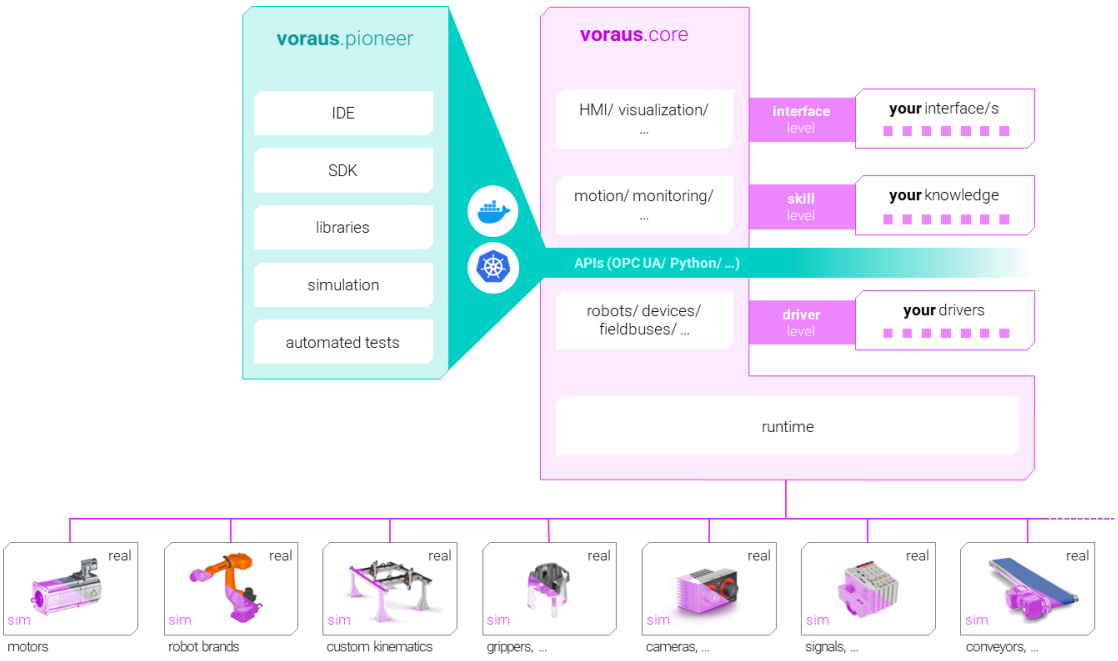

Figure 2: voraus.pioneer allows for holistic testing of the entire application in simulation and uses the same interfaces to the components as the real automation cell.

If these conditions are met, AI functionalities can be seamlessly integrated into the Industrial DevOps workflow (see figure 3 using the example of voraus.pioneer). Open interfaces based on IT standards allow for flexible and rapid integration of a wide variety of tools and libraries. The high level of automated test coverage detects errors at an early stage and drastically reduces risks during commissioning. As shown in section 4, this opens up completely new approaches to optimising KPIs: AI agents can suggest improvements, which are then subjected to the test pipeline and checked (just like the rest of the code). Now it becomes possible, for example, to easily and automatically optimise cycle times or space requirements at an early stage and secure competitive

advantages at “the push of a button”. Ultimately, this is an industrial implementation of an Agentic Feedback Loop (AFL) – i.e. the autonomous generation, testing, and correction of theories by AI agents.

Finally, there remains the question of how the (numerous) tests are created. Test specification and implementation are an inherent (albeit mostly unpopular) part of the software development process. In addition to the manual specification and implementation that is still common practice today, there are also approaches to automatically generate tests, for example from the requirements specification and from documents to be created during the specification/design phase (left side in the V-model). Unit tests are particularly promising in this regard, as the functionality is clearly described and there are numerous examples in the training data. The automatically generated tests are often combined with those specified by the programmer – as done in our case study in section 4.2.

As shown in the Industrial DevOps white paper, early detection of errors significantly reduces the cost of fixing them: For example, if errors are discovered during commissioning, the cost of fixing is 40 times higher than during the implementation phase (source: Jones, Caspers. Applied Software Measurement: Global Analysis of Productivity and Quality). This should be sufficient motivation (in addition to product liability) for a professional automated test management (e.g. using Jenkins) and high test coverage.

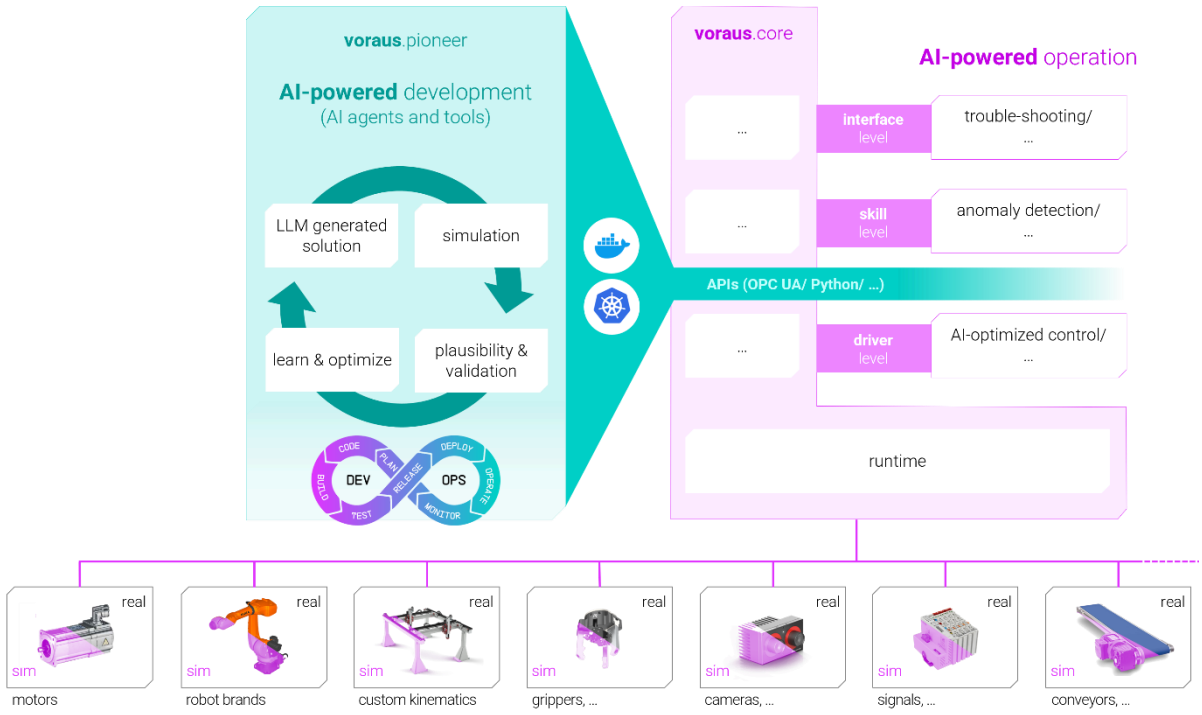

Figure 3: The voraus.pioneer enables an AI-driven software development process

4. Case Studies

The pick-and-place cell shown in figure 4, consisting of a robot with a vacuum gripper, a Roboception camera, a conveyor belt, three boxes filled with small plastic bags, and a touch display for placing orders, is controlled by the voraus.core. The entire application, including EtherCAT communication, runs centrally on a small IPC; no PLC is used. All components and the application itself are programmed in high-level language.

Figure 4: Pick-and-place cell with robot, conveyor belt, three boxes, and suction gripper

The simplified flow-chart for processing an order is given in figure 5: In a scan pose that does not obstruct the camera's view, the robot waits for an order (e.g. three bags from the middle box). The camera determines possible pick poses for the bags, the robot moves to one of those pick poses (via-points not shown here) and the vacuum gripper is activated. If the gripping fails, the process is repeated. If the gripping is successful, the robot moves to a deposit pose and the bag is placed on the conveyor belt. If the bag is lost during the motion (e.g. due to excessive acceleration forces), the gripping process is repeated, too. After successful placement on the conveyor belt, another bag is gripped if necessary until the entire order has been processed.

Based on this scenario, the following AI-supported optimisations are carried out:

- Increase of the pick rate (probability of successful gripping of a bag) by optimising the pick pose (section 4.1).
- Reduction of the cycle time through fully automatic code optimisation using generative AI (section 4.2).
- Reduction of the cycle time by interactive code adaptation using generative AI (section 4.3).
- Fully automated programming (section 4.4).

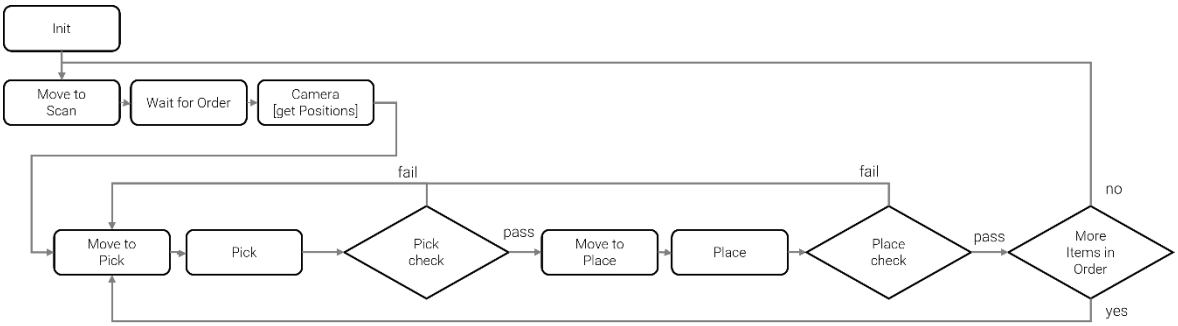

Figure 5: Simplified flow-chart of the pick-and-place application

4.1 Optimisation of Pick Pose

During cell operation, a log is kept of whether a pick was successful or not. The successful and unsuccessful picks per box are shown in figure 6 (green: successful, red: unsuccessful). Visualisation is set-up easily, e.g. using the open-source tool Grafana.

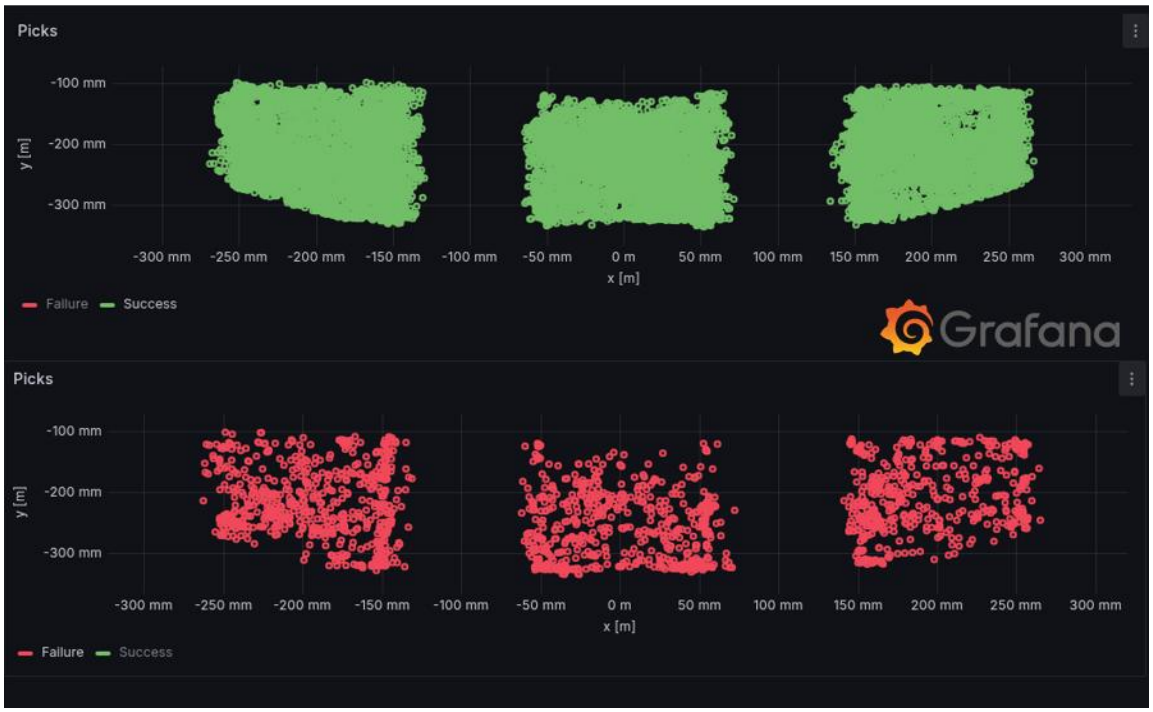

Figure 6: Successful (green) and unsuccessful picks (red) per box

This information is now used to train a model with PyTorch (a Python open-source programme library for machine learning) that calculates the probability of a successful pick for each pick pose suggested by the camera (up to ten poses are detected per box) at runtime. The bag with the highest probability is then grasped by the robot. This increases the pick rate (successful grasping of bags) from an initial 95 % to over 99 %. No additional sensors were integrated – only the information already available during operation was systematically evaluated and used to train an AI for regression.

4.2 Reducing Cycle Time Through Fully Automatic Code Optimisation

In this (and the following) example, one bag from each box is to be picked and placed on the conveyor belt. The aim of the optimization is to reduce the cycle time, which, like the increase of the pick rate, has direct impact on the economic efficiency of the system. The basis is a simulation of the cell including the application, which provides essential metrics for evaluation (test plan: duration of path segments, collisions, etc.), see figure 7. The application itself is programmed in high-level language (Python).

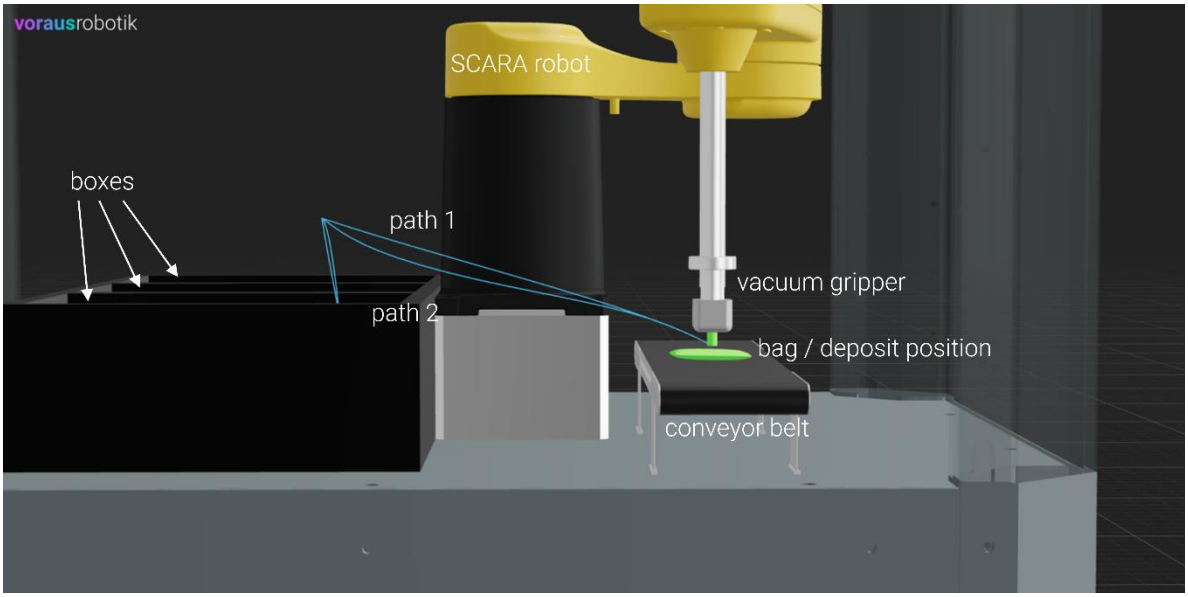

Figure 7: Simulation of pick-and-place cell

The GitHub CoPilot AI agent used Gemini 3 Pro Thinking as LLM for code generation. Through VSC (VSC: Visual Studio Code) integration, it had access to the application's original Python code, to the software documentation from voraus robotik, and, via MCP (model context protocol), to the InfluxDB database fed by the simulation (with information about collisions, cycle time, successful execution of the task, etc.). This ensures a clear linkage between simulation results and source code for each iteration of the optimisation.

The prompting included, among other things, the actual goal (minimising the time required for the pick-and-place task), which variables may be adjusted, and the creation of a test plan with success criteria.

In "Beast Mode," the AI agent is able to execute the generated code independently in a local DevContainer and subject it to the generated test plan based on the simulation. Each iteration of the optimisation took about one minute on a standard computer and included, among other things, starting the software containers, running the simulation, and reasoning. The total duration of the code optimisation was about 20 minutes.

After finishing the complete optimization run, the AI Claude (Anthropic) was asked what changes the AI agent had made to the code. The result is shown in figure 8: among other

things, blending between path segments (point 4) and synchronisation with the conveyor belt (point 6) were adjusted.

```
### Solution Implemented
Key changes in program_utils.py:
1. Increased HOVER_HEIGHT from 0.01m to 0.025m
2. Removed blending at critical pick/place hover positions (added .result() to wait
3. Increased linear movement speeds from 0.5x to 1.0x
4. Optimized blending percentages strategically:
   - 80% for initial approach to hover (safe open space)
   - 70% for OUT_OF_VIEW transitions (high-level safe movement)
   - 50% for post-pick return to hover (moderate safety)
5. Removed unnecessary sleep after pick operation
6. Optimized conveyor timing to 0.2s
```

Figure 8: Changes made by the AI agent to the pick-and-place application according to Claude

As result, the AI agent achieved a reduction of the cycle time from initially 13.85 seconds to 12.4 seconds, i.e. 10 %.

4.3 Reducing Cycle Time Through Interactive Code Optimisation

In contrast to the previous example, in this case the AI agent was not allowed to execute the generated source code itself. This procedure is similar to the usual co-pilot approach in programming and allows for better control of the code generation. In addition, it is possible to contribute one‘s own suggestions (and thus expert knowledge) to the optimisation process by adjusting the prompt. At the end of the approximately 40-minute interactive and iterative code optimisation, the cycle time was reduced to 11.6 seconds (instead of the initial 13.85 seconds). This corresponds to an improvement of over 16 % and exceeds the results of the fully autonomous approach described in section 4.2. The reason for this is likely due to the incorporation of expert knowledge into the optimisation process, which ultimately leads to more degrees of freedom in the optimisation process.

In both cases (section 4.2 and section 4.3), the optimised code, which was automatically tested in the simulation, was deployed to the real test cell without any changes.

4.4 Fully Automated Programming

Fully automatic code generation using generative AI is currently the "supreme discipline". Looking at the solutions available on the market, one gets the impression that they are either (still) very limited on the application side or tend to implement a natural language interface for parameterising existing processes. Even though these are important steps, existing solutions often fall short of their claim of universal validity. The current reasons for this are outlined in section 2.2 – in particular, the lacking of error- and edge-cases in (automated) testing should be emphasized again.

We were also unsuccessful in generating code automatically for the entire pick-and-place cell within a reasonable amount of time and with the quality required for industrial use – regardless of the LLM (ChatGPT, Claude, Gemini, and Grok). However, what worked very well was the generalisation of a small example: The pick process for one box was programmed and then generalised to the other boxes using AI. Figure 9 shows the manually written programme code (left) and the code generated by the AI for the middle box (right). The differences are highlighted in colour and illustrate that the LLM was able to understand and adapt the indexing of the boxes.

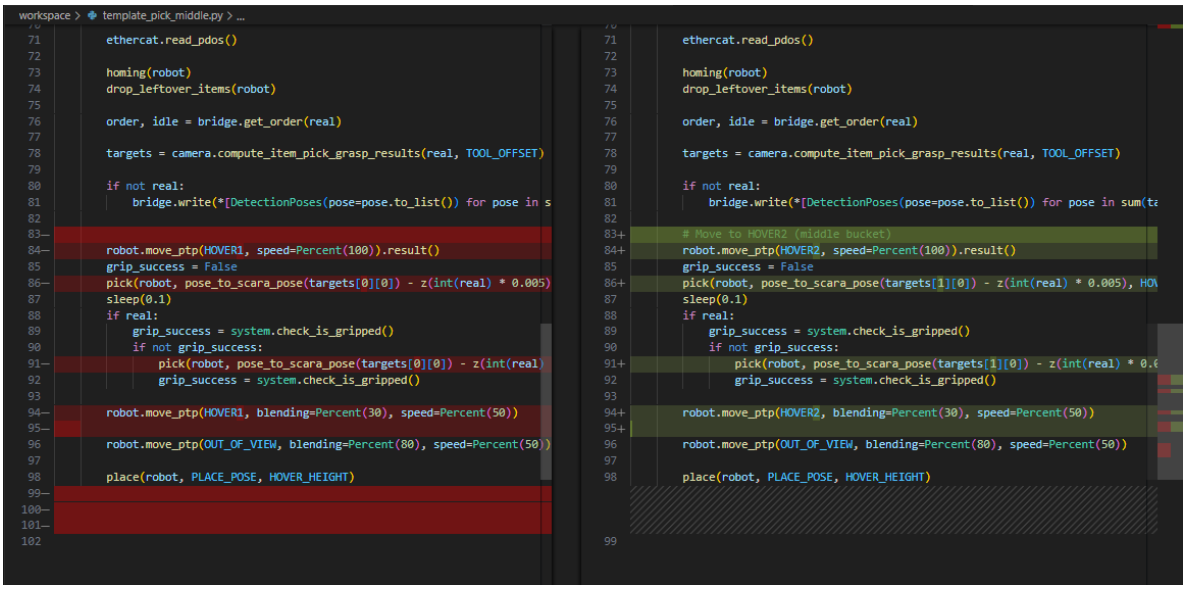

Figure 9: Initial example for one box (left) and generalization by AI (right)

After validation within the Industrial DevOps pipeline, this code was also directly and automatically applied to the real cell and executed.

5. Conclusion and Outlook

Based on the example of an industrial pick-and-place application (with commercially available, heterogeneous components), we demonstrated the practical benefits and economic added value that AI can already deliver today. This shows the power of AI agents for optimisation when we skilfully combine (generative) AI, simulation, and real cells. Let us briefly summarise the results: The pick rate was increased from 95 % to more than 99 %, exploiting data collected during operation. The use of generative AI decreased the cycle time for the pick-and-place application by more than 16 %. The latter integrated a simulation of the application and a test plan defined by the AI agent.

If we consider the time required for code optimisation in sections 4.2 and 4.3, even a conservative estimate is below an hour. The effort required to set up the AI agent is mainly due to the integration of simulation and test pipeline, which in turn are an integral part of the Industrial DevOps framework. If we add three hours for this extra work (again conservative estimation), the entire optimisation took half a day only. Extensive testing

before and during deployment is completely eliminated. To conclude, a reduction in time of up to 80 % compared to the current approach (manual optimisation, updating various control computers of the real cell, combined with extensive testing) is realistic.

The example of code generation in section 4.4 clearly underlines that the more specific the task can be defined, the less effort is required for prompting. This is the key to saving time: the goal is not necessarily the automatic programming of an entire application – it seems more efficient to generalise a manually created small example.

Both section 4.3 and section 4.4 show that the contribution of expert knowledge can significantly increase the performance of a purely AI-based approach. This is the current sweet spot of AI-supported development – experts provide their specific knowledge and AI relieves them of time-consuming routine tasks.

What other insights were gained? Setting up the test infrastructure (simulation and automated evaluation) is crucial for the successful deployment of AI agents. A consistent toolchain, such as that created through the Industrial DevOps approach, provides an excellent foundation for this and significantly reduces the effort. Programming in common high-level languages (Python, Rust, etc.) ensures that existing AI frameworks can be integrated easily and without problems. In every of the above presented cases, the (automatically tested) results can be transferred to the real cell with low risk and in the shortest possible time – thanks to a uniform code base.

To conclude, AI is already delivering real added value for industrial automation – beyond the current hype.

6. Company Information and Contact Details

Further information about our software platform and how we can support you in using AI in industrial automation can be found on our homepage http://www.vorausrobotik.com/en/. Our documentation with many examples is available here: https://docs.vorausrobotik.com/.

Parts of the case study are funded by the Federal Ministry for Economic Affairs and Energy (BMWE) on the basis of a resolution of the German Bundestag (project ALIGMO, KK5640702GR4).

Contact Details

voraus robotik GmbH | Carl-Buderus-Str. 7 | 30455 Hanover | Germany | vorausrobotik.com

Tobias Ortmaier | tobias.ortmaier@vorausrobotik.com