

Reuniting Data, Processes, and Forms: Petriflow Object-Centric Processes in the Age of AI

Gabriel Juhás^{*†}, Milan Mladoniczky^{*†}, Juraj Mažári[†], Tomáš Kováčik[†], Luboš Petrovič[†], Jakub Kovář[‡]

^{*}Faculty of Informatics

Pan-European University, Bratislava, Slovakia

gabriel.juhás@paneurouni.com, milan.mladoniczky@paneurouni.com

[†] NETGRIF, s.r.o., Bratislava, Slovakia

juhas@netgrif.com, mladoniczky@netgrif.com, mazari@netgrif.com, kovacic@netgrif.com, petrovic@netgrif.com

[‡] Lehrgebiet Programmiersysteme, FernUniversität, Hagen, Germany

jakub.kovar@fernuni-hagen.de

Abstract—For decades, enterprise information systems have been designed around separation into layers - data models are defined independently of business processes, and user interfaces are treated as an afterthought. Yet in business reality, these dimensions are inseparable: a business object is not a static collection of attributes, but an evolving entity whose identity, behavior, and representation form one holistic whole. This gap between how business understands reality and how IT models it has created what may be called the Great Divide of digital design. This paper explores Object-Centric Processes (OCP) as a paradigm that reunites data, processes, and forms into a single coherent model of business reality. By treating objects as both carriers of information and drivers of behavior, OCP provides a natural bridge between conceptual understanding and executable system design. We illustrate this approach through the Petriflow language, which captures the state, lifecycle, and interactions of business objects in an integrated, model-driven way.

Index Terms—low-code, workflow, object-centric processes, web applications, Petri nets.

I. INTRODUCTION – THE JOURNEY FROM CODE TO ABSTRACTION

Around 2006, after returning from Germany, I was building several information systems in Java. At that time, I worked with IBM’s Rational Software Architect, a professional IDE built upon the open-source Eclipse platform. The architecture we used - SQL databases, Hibernate mappings, Java Server Faces, and web applications running on WebSphere — was considered advanced. Yet, even then, the developer’s task remained largely the same: configure databases, set up servers, map entities, and build the user interface layer by layer.

I realized that despite all technological progress, software engineering still required an enormous amount of repetitive *plumbing*. It was clear that a higher-level language could—and should—abstract away these layers. A language that would not merely describe programs, but *processes*; that would let the developer focus on the logic of human and business workflows rather than infrastructure. This idea eventually became the foundation of **Petriflow**.

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the project No. 09I03-03-V04-00493

II. THE EVOLUTION OF SOFTWARE ENGINEERING

Since those early days, our industry has evolved dramatically. We have seen the rise of the Cloud, Platform-as-a-Service, Software-as-a-Service, and a dozens of frameworks—Spring, Angular, React—along with modern databases and query tools. These advances made it finally possible to realize a truly high-level, process-oriented language — something imagined two decades ago.

Yet much of full stack development remains bound to the same conceptual model: persistence, middleware, frontend, DevOps. We have built better tools, but not fundamentally different abstractions. The time has come for a *new paradigm* - for an **application language** that speaks directly the language of applications themselves [1].

III. THE PETRIFLOW VISION

Petriflow [2], [3] was envisioned as such a language: a *full-fledged low-code application language* that describes processes, tasks, and forms—not servers, databases, or frameworks. It builds upon the conceptual foundations of Petri nets but extends them into a practical programming model.

In this vision, the **Petriflow Virtual Machine (PVM)** [4] handles everything that full-stack developers now implement manually—data persistence, execution control, integration, and deployment. The programmer’s role shifts from coding technical layers to designing and orchestrating application logic in a human-readable, process-driven form.

Petriflow is not meant to replace developers—it is meant to **liberate** them. Just as C once liberated assembler programmers, and Java freed C developers from memory management, Petriflow aims to free future generations from the complexity of layered stacks.

IV. REQUIREMENTS ON A LOW-CODE LANGUAGE FOR OBJECT-CENTRIC PROCESSES

In this section, we discuss the role of Petri Nets [5], [6], [7] and Object-Centric Processes in recent development of Software Engineering. Namely, we illustrate how extended Petri nets can be used to define a low-code language for

modeling and developing software systems and web applications based on object-centric processes. Among others, we define key requirements for a low-code language for object-centric processes with forms. Such a language should enable to model object-centric processes consisting of data attributes, the life cycle of data attributes given by a workflow of tasks, and forms linked to these tasks consisting of subsets of data attributes. In addition, we specify requirements for handling events related to artifacts of object-centric processes, namely process instances, data fields, and tasks, resulting in a class of discrete event systems. We also discuss how roles [8] specifying which users can trigger events should be represented within such a language. Finally, we outline requirements for a query language capable of searching on object-centric processes and their tasks. We also provide a prototype of such a language based on Petri nets enriched by data attributes, called Petriflow [9]. Finally, we give an illustrative example of an application implemented using the presented low-code language Petriflow.

Before we define the requirements on the low-code language, let us specify the type of application that one should be able to develop using the low-code language.

The low-code language should primarily serve the digital transformation of business service processing. The applications in consideration should automate business processes consisting of process steps also called tasks or activities, which can be performed by users in different roles. The tasks in the processes could either be human tasks or machine tasks.

Typical applications include:

- Claim processing in insurance, where a process instance starts with the task/activity performed by a user, an insured person, that fills a form of the application for insurance with the details of the insurance claim as data attributes, followed by a task performed by an insurance adjuster.
- Loan Application in Banking, where customers initiate a loan application by providing personal details, financial data, and the requested loan amount through a standardized form. Subsequently, an underwriting officer reviews the application, performs a credit check, and assesses preliminary eligibility. If the application proceeds, a loan officer finalizes the terms of the loan and coordinates the disbursement, thus completing the process.
- Employee Onboarding in Human Resources. This process begins when a new hire provides personal information, banking details, and emergency contacts through an online form. The Human Resources department then initiates background checks, enrolls the employee in payroll systems, and registers relevant benefits. In parallel, the IT department assigns the necessary equipment and creates user accounts. The manager reviews the onboarding progress and confirms successful completion before closing the process instance.
- Purchase Order Management in Supply Chain. Initially, a department initiates a purchase order by specifying item descriptions, quantities, and supplier information in

a standardized form. Next, a purchasing manager reviews the request, adjusts quantities or suppliers if necessary, and approves the order. The warehouse team updates the process as items are received and the finance team subsequently processes payment, concluding the order cycle.

- Permit Approval in Government Services. Government agencies often rely on process-based applications to handle permit approvals. An applicant, such as a property owner or business representative, completes a form detailing the scope of the permit (e.g., building permit), along with any required supporting documents. A review team then assesses compliance with local regulations, and officials may perform on-site evaluations, updating the process with their findings. Once all requirements are met, the responsible authority grants the permit, thus concluding the approval process.

The common feature of all above-mentioned applications is that they combine data management with process management. For example, one could consider a loan to be a data object with different attributes, such as the amount of money borrowed by the customer of a bank, customer name, surname and address, the monthly income of the customer applying, a Boolean attribute stating whether the customer is eligible to get the loan based on attributes of loan amount, the loan length, and the monthly income. However, the loan can also be considered to be a process as described above, starting with the task of loan application performed by a customer, followed by other tasks performed either by bank employees, i.e., by human users or by bank systems, i.e., by a machine user. The human tasks, such as filling and submitting the application for a loan, require a graphical user interface, where users can enter the values for a subset of loan attributes. Traditionally, such user interface consist of a web form containing the fields for the loan attributes. Machine task, on the other hand, require an endpoint they can access over the inter- or intranet to provide and read the required loan attributes.

When modeling and implementing such applications, traditional approaches of software engineering, such as model driven development, suggest to model loan data and their relations separately from the process describing the loan service. Instead, we propose to combine the data and the workflow of the loan into one object-centric process, consisting of set of data attributes together with a workflow defining the life-cycle of the data attributes. Such workflow process consists of a flow of tasks, where tasks are associated with subsets of data attributes of the object, together with an information, about how they can be approached by a user performing the task (whether they should be entered by a user, just displayed to a user, recomputed whenever other attributed are changed by a user, etc.) resulting in forms as user interface.

A. Building blocks of the low-code language

In this section, we discuss entities that are artifacts or building blocks of the low-code language based on object-centric processes with forms. These building blocks should

have a defined notion of a state. Because of the reactive nature of the application to be developed using the low-code language, one should define a set of events for each entity, that change the state of the entity. Mathematically, the state-event model of an entity should be defined by a suitable formalism for discrete event systems, such as automata, state machines or Petri nets.

Although we want to remain as abstract as possible, the low-code language should serve for the development of real enterprise applications, where user interface given by forms is accessed by users via a web browser.

As it follows from the title, the basic type of entities of the low-code language, analogous to a class in object-oriented programming or a table in relational databases, is an object-centric process. Then, instances of object-centric processes correspond to objects of classes or records in tables.

An object-centric process should consist of:

- 1) The data part formed by data variables or attributes of a given type, where available primitive types should include text, number, and boolean similar to low-level programming languages, also more complex predefined types, such as file, date or different types of selectable collection, such as enumeration or a multi-choice, and types used as references, such as case ref and task ref. One should also define at least two basic events on data variables, one for reading the data variables value and one for changing the data variables value. Similarly to object oriented programming languages, one can define static data variables shared by all instances, i.e., having process scope and dynamic data variables created for each instance, i.e. having instance scope.
- 2) A workflow consisting of the flow of tasks representing the life-cycle of the object-centric process. A workflow is modeled as a structured sequence of tasks that governs how an object transitions through various states over its life-cycle. Each task is triggered when predefined conditions are met, producing changes in data attributes or the object's status. This approach supports parallel execution and synchronization of multiple tasks, handles branching into different process paths, and captures the overall progression from initiation to completion. By enforcing clear rules for when and how tasks activate, the workflow precisely defines the life cycle of the object-centric process in a way that accommodates both simple and complex behavioral patterns.
- 3) Forms associated with tasks consisting of a subset of data attributes accessible by a user with permissions to perform the task, also called data references. Depending on the state of data references, events can be triggered by a user on given data references. Forms are typically associated with specific tasks and contain only those data attributes that a user is authorized to view or modify. In practice, this means that when a user is assigned a task, the system provides them with a tailored interface reflecting only the referenced data variables they need to complete their portion of the workflow. As these

data attributes are changed, whether through user input, computation, or external services, the system updates its current state in real-time. In turn, transitions in the state of these data references can trigger predefined events, such as notifications, automated validations, or the activation of subsequent tasks. This structure ensures that each user interacts solely with the pertinent subset of the information required for their assigned task, thereby maintaining data integrity and preserving clear boundaries of responsibility within the object-centric process.

- 4) Events defined on specific parts of the process. Events can be defined for data attributes, as set event to change the value of the data attribute or get event to access the value of the attribute. Different events can be also defined for tasks, such assign event assigning the task to a user, cancel event canceling the assignment, and finish event finishing the assigned task. Specific events can also be defined for processes to activate (deploy) or deactivate a process and for process instances to create or delete instances of the process.
- 5) List of users called roles. Basically, events can be triggered by users with permissions, where users can be humans or machines. For this purpose, one can use roles, which are basically special data attributes of processes consisting of lists of users. They can be associated with events via role references. Roles define who has permissions to trigger the associated events.
- 6) Actions as reactions to events. Actions are anonymous functions that define reactions to all types of events. In actions one can search for users, instances of processes and their tasks using queries. In actions one can also trigger events on processes, their instances and data attributes and call internal and external functions.

V. OBJECT-CENTRIC PROCESSES IN PETRIFLOW

In this section we provide informal definitions for basic entities of a low-code language Petriflow for object-centric processes.

As mentioned above, object-centric process consists of a set of Data attributes together with their life-cycle in form of a Workflows with explicit states and a set of Tasks with Forms.

A. Process lifecycle and versioning

Multiple versions of the process can exist in the application at the same time to support long running processes and changes in real world that occur during their lifecycle that forces the creation of new version of the process. There is also need to limit the ability to create instances of old processes, so every process needs to have state called *activated*, that enables or disables the creation of instances of that event. Activation of process can be executed directly after the new version of process is created or scheduled to happen at certain time in the future.

B. Workflows

As a workflow model, Petriflow uses place/transition Petri nets [10], [11], [12] enriched by reset arcs [13], inhibitor arcs [14] and read arcs [15]. Transitions of Petri nets represent tasks of workflow models. In addition to constant values Petriflow allows to use dynamic expressions with integer return type as arc weight, i.e., arcs with variable weights or shortly variable arcs [16].

C. Cases

Instances of Object-centric processes in Petriflow are called cases. Each case contains a copy of the parent process, including workflow and data variables. Dynamic values of attributes of process elements are evaluated during the instantiation of the case. Static values of attributes of process elements are cloned during the instantiation of the case.

D. Tasks

Task represents an activity that Users (human or machine) performs. Tasks are associated with subset of data attributes of the process by Data references also called dataRefs. By performing a task by a user we mean that the user triggers Get task data event or Set task data event of its data references. Task can be disabled or enabled based on state of the workflow, i.e., by the marking of the underlying Petri net. If a task was disabled and is becoming enabled an Enable task event of the task is triggered by the system. If a task was enabled and is becoming disabled an Disable task event of the task is triggered by the system. If a task is enabled it means that it can be executed by a user. Execution of task starts by triggering Assign task event that assign a user to given task and ends by triggering Finish task event. Petriflow uses so called interval semantics of transition firings [14] (also called first-consume-then-produce), where Assign task event of task represented by a transition consumes tokens and Finish task event produces tokens. Whether a user can assign a task to himself or a different user is determined by Role references. By triggering a View task event for disabled, enabled or assigned task the task is displayed for the user with role given by role reference. Similarly role references are also used for finish events. If a task is assigned to a user it also can be cancelled by triggering Cancel task event, resulting in returning of tokens consumed by Assign task event.

E. Forms

Forms with different Layouts, Data references, and user interface components can be defined in the Object-centric processes in Petriflow. Each task can reference one form with it. A form can be referenced and reused in many tasks. Forms serve as interface for human interaction with associated data fields of the task.

1) *Layouts*: Layout defines the size, position, and alignment of data fields. Petriflow uses two types of layout, *flex* and *grid*. Flex is a one directional layout that places its items in a row or column to fill available space. Grid is a two dimensional layout that places its items in a grid with a given row and

column size. The layout item can be either a data field, an empty space, or another layout of any type. A combination of flex and grid allows to design complex layouts for any use case.

F. Identifiers

Each element in the Object-centric processes in Petriflow needs to be identified by a unique value - identifier. The identifier must be unique in respect to the elements of the process and all its ancestors through inheritance. The identifier must be a valid identifier in the platform programming language. This simplifies Actions and allows to directly reference objects and entities of the current context, e.g. data variables of the case.

G. Expressions

Expressions are one-line anonymous functions that are evaluated during the runtime when needed. Expressions can be used to initialize data variable with dynamic values, serve as dynamic weight of arcs in the Workflows, and in many other situations.

Example: initialize the value of the data variable `due_date` to 30 days in the future from the current date.

H. Functions

Similarly to object-oriented programming, functions are named, and can have arbitrary number of arguments and a value. Functions have a scope that determines in which context they can be called. Functions with scope process can be called in all events of cases of that process. Functions with scope application can be called in all events of cases belonging to that application.

I. Data variables

Data variables represent all attributes of a process instance called case during its life-cycle. Data variables associated to workflow tasks define data fields and create task forms.

J. Data attributes

Each data variable has its type and a set of attributes that further specify its purpose in the process. The core attributes that are always present in the set are: title, description, placeholder, initial value, list of validations, and default component for user interface. At present, there are following available datatypes in Petriflow: number, text, enumeration, multichoice, boolean, date, date time, file, file list, user, role (user list), button, task ref (list of task identifiers), case ref (list of case identifiers), `i18n` (text with possible multiple values with a default value and possible translations of the default value to different languages). For data variables of type enumeration and multichoice there is an attribute `options`, which is a map (set of pairs string key and string value to select one or multiple options as a value of an enumeration variable or multichoice variable).

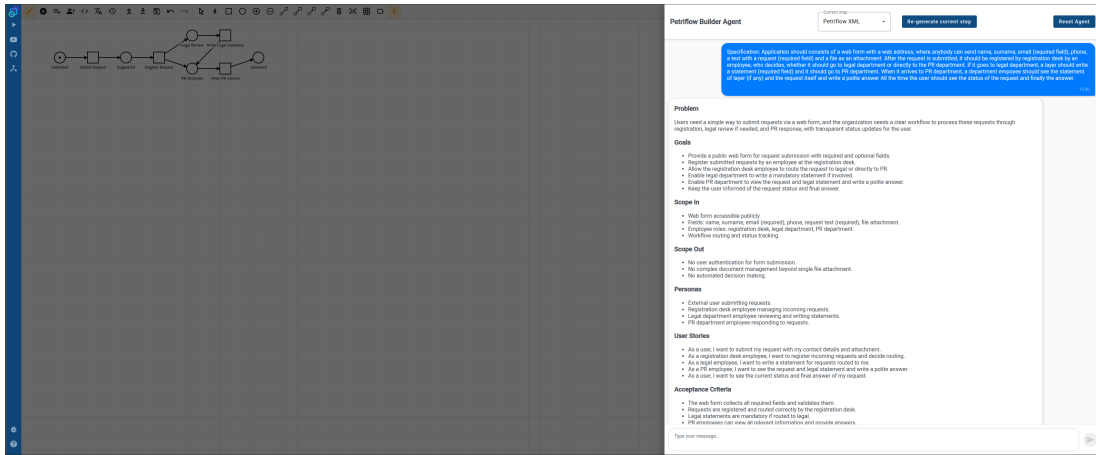


Fig. 1. Petriflow Builder Agent with simple specification in natural language

K. Data references

Data variables are associated with Forms using data references. Data references has both its own identifier and the identifier of the referenced data variable. Data references allow to specify if a user needs to provide a value for the data variable using the *required* attribute. The *behaviour* attribute defines if the graphical input field is visible to the user and whether it is editable.

L. Validations

Validations are used to ensure only valid data are persisted into the database. Validation can be run on server or client side or both. Custom validations can be created and used in processes. If a validation fails (returns false) associated error message is logged (server side) and/or showed to the user (client side). Validations can have 0..N arguments to support validity based on a set of dynamic values, for example a date field value is valid if it is within 30 day range from the value of another date field.

M. Actions

As mentioned, actions are anonymous functions that define reactions to all types of . In actions, events can be triggered and external functions can be called. Since events can change the state of the process, actions need to be attached to execution phase before the event, i.e. pre-phase, or after the event, i.e. post-phase.

Actions can have different context available to them depending on the type and phase of event it is attached to. For example action of Create case in the pre-phase does not have access to the data variables of the case because they have not been created yet.

N. Properties

In many real-life use cases, additional information need to be stored with the process or parts of it. Properties can be used to store them in the form of a key-value store (string:string).

VI. USERS AND ROLES

User management and role-based access control are integral part of Petriflow.

A. Users

All entities interacting with Petriflow Virtual Machine are collectively called users. From the point of view of a process, it does not matter if a task is executed by human being, by robot, or by another application via integrations.

B. Roles

Roles are sets of Users. They can be associated with sets of Events by Role references also called roleRefs. Role references allow (positive permission) or forbid (negative permission) the triggering of an associated event by its role users. Roles are special data attributes and therefore can be static or dynamic determining permissions for process (process events and events of all cases) or a single case.

1) *Case Roles*: Case roles are lists of users specific for a given instance and therefore can be associated only with events of the instance. Positive permission example: only the thesis supervisor can assign student to the thesis. Negative permission example: the thesis supervisor cannot be the thesis opponent.

2) *Process Roles*: Process roles can be associated with process events and give permissions for each instance of that process. Positive permission example: only loan officers can approve mortgage applications. Negative permission example: interns cannot assign tasks to themselves.

C. Role references

Role references associates Roles with Events. For each event they define with true/false value if users of that role are allowed (true) or forbidden (false) to trigger the event.

VII. EVENTS

Dynamic behaviour of Petriflow object-centric processes and their instances are represented by events.

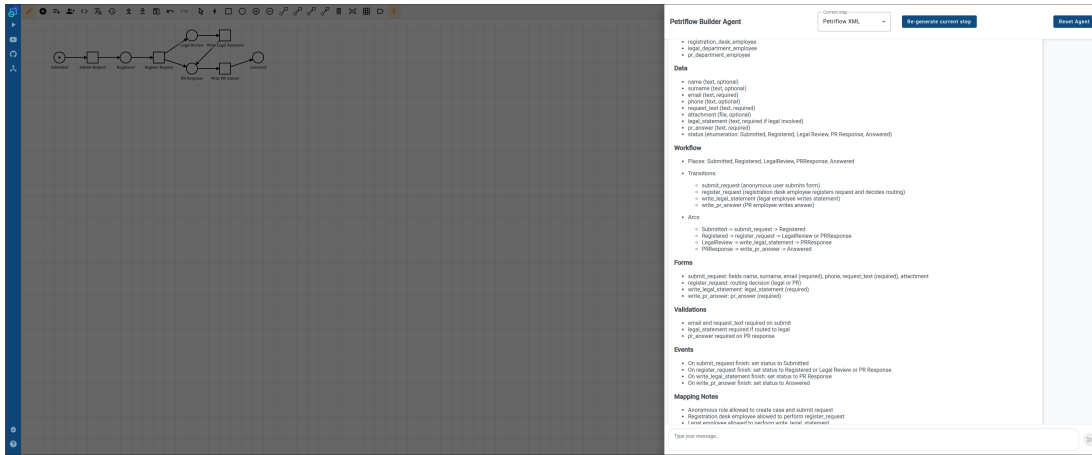


Fig. 2. Petriflow Builder Agent with structured technical specification generated from natural language specification

A. Process events

Process events are defined by the lifecycle of Object-centric processes in Petriflow.

1) *Create process*: Create process event is triggered when a new process is created or uploaded in the application or if a new version of existing process is created or uploaded. Example: create process event enables to run migration scripts on existing instances of previous version of the process.

2) *Delete process*: Delete process event is triggered when an existing version of a process is deleted. Deleting process also deletes all cases of that process and triggers Delete case on those instances.

3) *View process*: The view event allows users to view detailed information of the given process and allows it to be searched.

4) *Activate process*: Activate event is triggered when the process is activated and new instances are enabled to be created.

5) *Deactivate process*: Deactivate event is triggered when the process is deactivated and new instances are forbidden to be created.

B. Case events

Case events are defined by the lifecycle of instances of Object-centric processes in Petriflow and are triggered by creating, deleting, and searching of process instances.

1) *Create case*: Create event is triggered when a new case of a Object-centric processes in Petriflow is instantiated.

2) *Delete case*: Delete event is triggered when an existing case of a Object-centric processes in Petriflow is deleted.

3) *View case*: View event is triggered when a case is being viewed. View event allows to restrict which users are allowed to view the case. This also means that the case will not appear in ?? searches. View event also provides immediate data fields of the case.

C. Task events

Task events are defined by the lifecycle of tasks of instances of Object-centric processes in Petriflow.

1) *Assign task*: Assign event marks the start of execution of a task. Assign event assigns the user to the task and change the state of the workflow by consuming tokens from input places.

2) *Cancel task*: Cancel event stops the execution of the task. Cancel event removes the assigned user and rollback the change of workflow state by returning as many tokens to input places as assign event consumed. Any changes made to data variables during the execution are not rollbacked by default.

3) *Finish task*: Finish event concludes the execution of the task. First it runs validations and checks required data fields for values and if any of them fails the finish event stops. After that the finish event removes the assigned user and change the state of the workflow by producing tokens to output places.

4) *Enable task*: Enable event is only triggered by the change of state of the workflow when the task becomes enabled. Enable event is important in cases where you need to react when part of the workflow enters a certain state from any previous state, e.g. on OR-join.

5) *Disable task*: Disable event is only triggered by the change of state of the workflow when the task becomes disabled.

6) *View task*: View event is triggered when a task is being viewed. View event allows to restrict which users are allowed to view the task. This also means that the task will not appear in ?? searches. View event also provides immediate data fields of the task.

7) *Set task data*: Set data event is triggered when attributes of one or multiple data references are changed. This also triggers set event of referenced data variable.

8) *Get task data*: Get data event is triggered when the content of the task (data form) is read. This also triggers get event on all referenced data fields.

D. Data events

Data events are connected with data variables of the process and are triggered by reading values and attributes of data variables and by changing values or attributes of data variables.

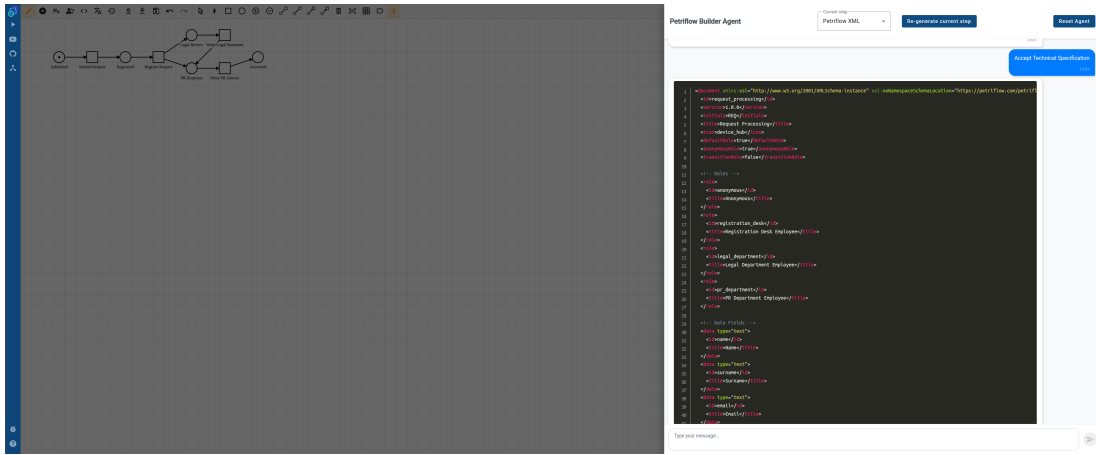


Fig. 3. Petriflow Builder Agent with source code in Petriflow generated from structured technical specification

Difference between Set task data and Set data field, and Get task data and Get data field is that task events are only triggered on data fields of that task and data events are triggered always when data variable is changed or read. This allows developers to react on changes made in specific task and for example change visibility of data references in task forms.

1) *Set data field*: Set data event is triggered when the value or attributes of the data variable are changed.

2) *Get data field*: Get data event is triggered when the value and attributes of the data variable is read.

E. Queries

Petriflow also provides functions called queries for searching users, cases and their tasks. These functions have predicates as parameters. A search function return all users, cases or tasks that fulfill the predicate that is the function parameter. Such queries can be called in actions.

VIII. PETRIFLOW VIRTUAL MACHINE

Today, Netgrif Application Engine provide an implementations of **Petriflow Virtual Machine** that provides full technology stack for Petriflow processes and allows developers to build and run enterprise applications based on object-centric processes in a cloud-native environment.

IX. LOOKING AHEAD – A NEW GENERATION OF DEVELOPERS

Every few decades, a new generation of programming languages redefines how we think. Assembler gave way to C. C gave way to Java. Now, the age of Java and multi-layered architectures is nearing its end.

The next generation of developers will not think in terms of controllers, APIs, or pipelines—they will think in terms of *applications, processes, and interactions*. They will focus less on the mechanics of implementation and more on expressing intent: what the system should achieve, not how it should be wired together.

At the same time, **Artificial Intelligence (AI)** enters this evolution not as a replacement for human creativity, but as the next natural instrument of abstraction. Just as calculators once replaced manual arithmetic—freeing mathematicians to focus on reasoning and insight—AI-assisted *vibe coding* will replace much of manual programming, allowing developers to concentrate on modeling meaning, logic, and process semantics rather than syntactic details.

The Petriflow language, by design, supports this paradigm. Because Petriflow operates on a higher level of abstraction than lower-level technologies such as HTML, JavaScript, Java, or SQL, it requires significantly less syntactic detail to express the same functionality. A single data form or process modeled in Petriflow can be described with only a fraction of the code that would be required in full-stack frameworks. By abstracting away communication and data flow between the user interface, application layer, and database, Petriflow not only simplifies development for humans—it also makes application generation easier and cheaper for AI systems.

This is a crucial insight: since AI models work by generating structured text, smaller and more regular languages such as Petriflow are inherently more suitable targets for AI generation. The reduced search space, the formal structure of object-centric processes, and the deterministic semantics of Petriflow make it possible for AI systems to generate more complex and consistent applications than in full-stack technologies. Moreover, the resulting models are far more readable and comprehensible to human developers, enabling an iterative, collaborative development process where AI and human expertise complement each other.

To explore this synergy, an **AI Assistant** has been integrated directly into the **Netgrif Application Builder (NAB)**, the development environment for Petriflow. This assistant enables developers to describe desired functionalities in natural language and obtain automatically generated Petriflow models—complete with data objects, workflows and forms — that can then be refined or extended manually. Such integration demonstrates how human and machine intelligence can co-

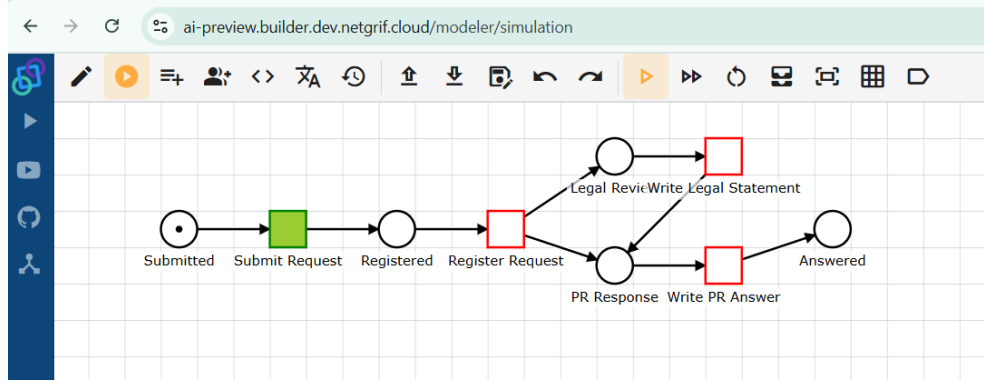


Fig. 4. Generated workflow of Petriflow process

create applications: AI accelerates prototyping, while humans ensure correctness, alignment, and creativity.

X. ILLUSTRATIVE EXAMPLE

In this section, we will briefly describe an illustrative example of a Petriflow process generated in Netgrif Application Builder with the help of AI assistant.

The Petriflow process was specified by the following text, that served as an input to Petriflow Builder Agent as demonstrated in Figure 1:

Application should consist of a web form with a web address, where anybody can send name, surname, email (required field), phone, a text with a request (required field) and a file as an attachment.

After the request is submitted, it should be registered by registration desk by an employee, who decides, whether it should go to legal department or directly to the PR department.

If it goes to legal department, a lawyer should write a statement (required field) and it should go to PR.

When it arrives to PR department, a department employee should see the statement of lawyer (if any) and the request itself and write a polite answer.

All the time the user should see the status of the request and finally the answer.

As you can see in the specification text, there are some grammatical errors to test how the assistant will deal with them, such as *em layer* instead of *lawyer* or *empoyee* instead of *employee*. In addition, specification is very brief and not very exact, not stating what are tasks or roles.

The first output from Petriflow Builder Agent is a technical specification, containing information about proposed roles, data fields, tasks, and forms, as it is demonstrated in Figure 2.

Here, a user can repetitively send feedback to improve the generated technical specification.

In this illustrative example, we just confirmed the first suggested technical specification without any feedback.

As a result, Petriflow Builder Agent provided source code of Petriflow process and displayed it in the Builder, as it is demonstrated in Figure 3.

Despite the fact, that the specification contained grammatical errors, it was definitely not precise enough and we did not do any iteration, we got syntactically valid source code with data fields, tasks, roles and forms displayed in Figure 4.

There were some minor manual extensions of the generated model, namely we rearranged placement of elements of workflow, we had to add some missing field on some forms, for example name and surname on the form of task *Write PR Answer* and we had to add a task for display of status. The only major addition to get correct process was addition of two number data fields, namely *toLegal* and *toPR* and an action on pre-phase of finish event of the task *Register Request*. In this action we just used one if-else command where in case that generated enumeration with two options *Legal Department* and *PR Department* had chosen option *Legal Department*, then we set *toLegal* to one and *toPR* to zero, otherwise we set *toLegal* to zero and *toPR* to one. By this manual change we got the correct final Petriflow process displayed in Figure 5.

XI. CONCLUSION – BECOMING THE CHANGE

In this paper, we have defined the fundamental **requirements for a low-code language** capable of expressing object-centric processes that unify data, processes, and forms within a single coherent model. We have also introduced an informal **definition of the Petriflow language**, based on extended Petri nets, illustrating how it enables modeling of object lifecycles, tasks, and user interactions through a structured, declarative syntax. Finally, we have discussed the emerging **synergy between Petriflow and Artificial Intelligence**, showing that the language's abstraction and compactness make it both human-readable and AI-generatable, allowing more efficient collaboration between human and machine creativity.

However, as with every technological leap, we are again witnessing a new *Great Divide*. On one side stand the enthusiasts who see AI as a full replacement for human intelligence, and on the other, the skeptics who reject it entirely. This polarization arises from a misunderstanding of AI's non-deterministic nature—it does not generate code predictably like a compiler but probabilistically, with variation and limited transparency. Rather than fearing this property, we should embrace it as a feature. AI should be integrated as a **creative**

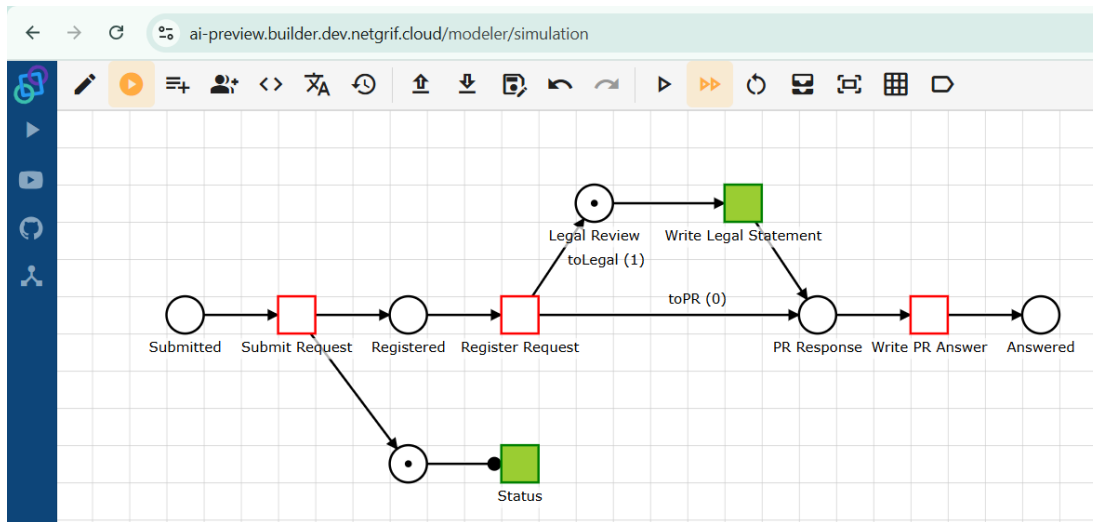


Fig. 5. Generated workflow of Petriflow process

collaborator, capable of proposing design alternatives and accelerating early development phases. When grounded on deterministic foundations such as **Petriflow**, which provides a rigorous and well-defined execution model, AI can safely operate as a productive partner. In this synergy, the generative power of AI complements the structural precision of model-based engineering.

The consequences of this shift are profound. By working at a higher level of abstraction, AI systems can generate meaningful applications more effectively in languages like Petriflow than in full-stack technologies. Moreover, the output is inherently more readable and structured, allowing human developers to review, understand, and refine AI-generated models rather than treating them as opaque code artifacts.

Although AI can make application development more effective, just as we continue to teach children to perform arithmetic by hand — even in a world filled with calculators — we must continue to teach students the **fundamentals of computer science and programming**. Understanding algorithms, data structures, and logic remains essential because it builds the cognitive framework that allows future engineers to reason about systems, not merely operate tools. Only by mastering the *why* behind computation can they meaningfully guide and supervise the *how* performed by AI.

The transformation from full stack technologies to more abstract low-code languages combined with AI code generation is not distant; it has already begun. We are witnessing the birth of a new era in software engineering — where data, processes, and forms converge, and where AI becomes a collaborator in design. The coming decade will belong to those who can bridge human understanding and computational execution. And perhaps, more than observing this change — **we are helping to create it.**

REFERENCES

[1] G. Juhás, L. Molnár, A. Juhásová, M. Ondrišová, M. Mladoniczky, and T. Kováčik, “Low-code platforms and languages: the future of

software development,” in *Proc. 20th Anniversary of IEEE International Conference on Emerging eLearning Technologies and Applications, ICETA 2022*. IEEE, 2022, pp. 286–293.

[2] A. Juhásová, G. Juhás, L. Molnár, M. Ondrišová, J. Mažári, and M. Mladoniczky, “It induced innovations: Digital transformation and process automation,” in *Proc. ICETA 2019 - 17th IEEE International Conference on Emerging eLearning Technologies and Applications*. IEEE, 2019, pp. 322–329.

[3] G. Juhás, T. Kováčik, J. Kovář, M. Kranec, and Petrovič, “Petriflow language and netgrif application builder,” vol. 2973, pp. 171–175, 2021.

[4] —, “Netgrif application engine,” in *BPM (PhD/Demos)*, ser. CEUR Workshop Proceedings, vol. 2973. CEUR-WS.org, 2021, pp. 166–170.

[5] J. Desel and W. Reisig, “The concepts of petri nets,” *Softw. Syst. Model.*, vol. 14, no. 2, pp. 669–683, 2015.

[6] J. Desel, “Process modeling using petri nets,” in *Process-Aware Information Systems*. Wiley, 2005, pp. 147–177.

[7] W. M. P. van der Aalst and K. M. van Hee, *Workflow Management: Models, Methods, and Systems*, ser. Cooperative information systems. MIT Press, 2002.

[8] R. Bergenthum, J. Desel, and S. Mauser, “Workflow nets with roles,” in *EMISA*, ser. LNI, vol. P-190. GI, 2011, pp. 65–78.

[9] G. Juhás, A. Juhásová, and L. Petrovič, “Low-code languages in it education: Integrating theory and practice,” in *Prof. ICETA 2023 - 21st Year of International Conference on Emerging eLearning Technologies and Applications*. IEEE, 2023, pp. 249–257.

[10] J. Desel and G. Juhás, ““what is a petri net?”,” in *Unifying Petri Nets*, ser. Lecture Notes in Computer Science, vol. 2128. Springer, 2001, pp. 1–25.

[11] W. M. P. van der Aalst, J. Desel, and A. Oberweis, Eds., *Business Process Management, Models, Techniques, and Empirical Studies*, ser. Lecture Notes in Computer Science, vol. 1806. Springer, 2000.

[12] J. Desel, B. Pernici, and M. Weske, Eds., *Business Process Management: Second International Conference, BPM 2004, Potsdam, Germany, June 17-18, 2004. Proceedings*, ser. Lecture Notes in Computer Science, vol. 3080. Springer, 2004.

[13] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn, “Soundness of workflow nets with reset arcs,” *Trans. Petri Nets Other Model. Concurr.*, vol. 3, pp. 50–70, 2009.

[14] M. A. Alqarni and R. Janicki, “On interval process semantics of petri nets with inhibitor arcs,” in *Petri Nets*, ser. Lecture Notes in Computer Science, vol. 9115. Springer, 2015, pp. 77–97.

[15] W. Vogler, “Partial order semantics and read arcs,” *Theor. Comput. Sci.*, vol. 286, no. 1, pp. 33–63, 2002.

[16] R. Valk, “Self-modifying nets, a natural extension of petri nets,” in *ICALP*, ser. Lecture Notes in Computer Science, vol. 62. Springer, 1978, pp. 464–476.