

## Article

# Developing Web-Based Process Management with Automatic Code Generation

Burak Uyanık <sup>\*,†</sup>  and Ahmet Sayar <sup>†</sup> 

Kocaeli University, Computer Engineering, Faculty of Engineering, Umuttepe Campus, İzmit 41380, Türkiye; ahmet.sayar@kocaeli.edu.tr

\* Correspondence: burak.uyanik@tei.com.tr; Tel.: +90-538-7478374

† These authors contributed equally to this work.

**Abstract:** Automated code generation and process flow management are central to web-based application development today. This database-centric approach targets the form and process management challenges faced by corporate companies. It minimizes the time losses caused by managing hundreds of forms and processes, especially in large companies. Shortening development times, optimizing user interaction, and simplifying the code are critical advantages offered by this methodology. These low-code systems accelerate development, allowing organizations to adapt to the market quickly. This approach simplifies the development process with drag-and-drop features and enables developers to produce more effective solutions with less code. Automatic code generation with flow diagrams allows one to manage inter-page interactions and processes more intuitively. The interactive Process Design Editor developed in this study makes code generation more user-friendly and accessible. The case study results show that a 98.68% improvement in development processes, a 95.84% improvement in test conditions, and a 36.01% improvement in code size were achieved with this system. In conclusion, automated code generation and process flow management represent a significant evolution in web application development processes. This methodology both shortens development times and improves code quality. In the future, the demand for these technologies is expected to increase even more.

**Keywords:** automated code generation; process flow management; web-based application development; dynamic process flow optimization; low-code platforms; flow-based programming paradigms



**Citation:** Uyanık B.; Sayar A. Developing Web-Based Process Management with Automatic Code Generation. *Appl. Sci.* **2023**, *13*, 11737. <https://doi.org/10.3390/app132111737>

Academic Editor: Luis Javier Garcia Villalba

Received: 24 September 2023

Revised: 17 October 2023

Accepted: 22 October 2023

Published: 26 October 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

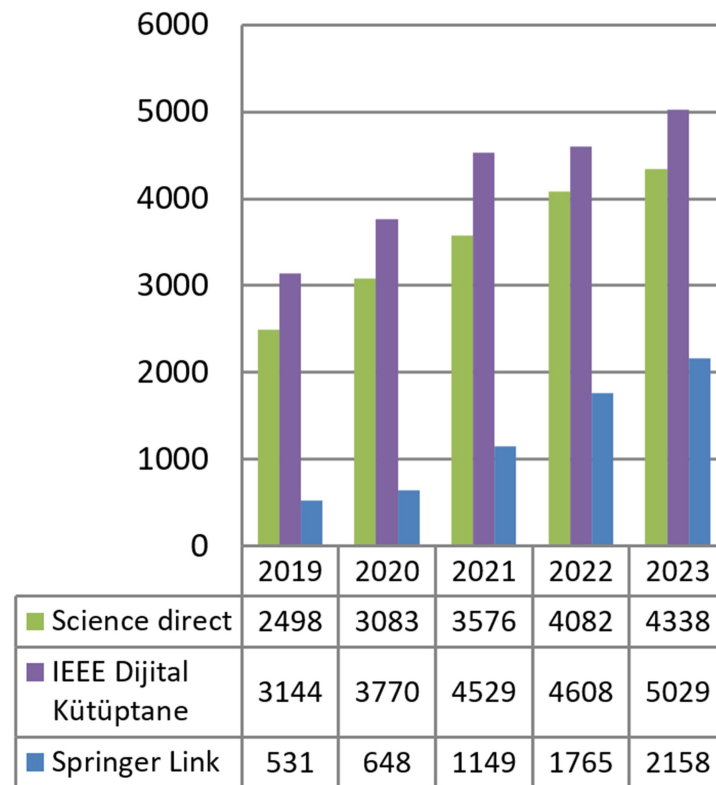
## 1. Introduction

Automated code generation has witnessed remarkable advances in software engineering in the last five years. The main goal of this technology is to improve coding quality and consistency in the software development process, thereby optimizing the developer's productivity and speeding up the development process. The concept of automatic code generation has been the focus of growing interest in computer science and software engineering. Prestigious academic platforms such as IEEE, Science Direct, and Springer Link have critically disseminated these advancements. As a result of our literature review, we have observed a sharp increase in studies on automatic code generation between 2019 and 2023 (Figure 1). Recently, the software engineering paradigm has seen in-depth analyses of techniques for the efficient and fast development of web applications. The literature on automated code generation suggests numerous methodologies and integration tools. These methodologies and tools zero in on the critical success criteria of software engineering, such as cost-effectiveness, code quality, and efficiency.

The advantages of automated code generation are geared towards optimizing software development processes:

1. Thanks to the modular and scalable code structure, a single developer or team can manage the user interface and server-side development processes more effectively.

2. Using advanced templates and libraries reduces the complexity of user interface design and enables rapid prototyping.
3. Predefined code templates optimize reuse and help avoid potential errors.
4. Standardized code structures and nomenclature increase consistency and traceability in code generation processes.



**Figure 1.** Number of studies on automatic code production.

However, the disadvantages of automatic code generation should not be overlooked:

1. During code generation, complex code blocks automatically generated by the system can make maintenance and update processes difficult.
2. Automatic code generation built around specific standards and templates may restrict customization.
3. Automated code generation systems with intricacies and in-depth details can be complex for novice software engineers; therefore, additional investments in training and adaptation processes may be required.

This study uniquely integrates two separate systems: automatic code generation and business process management. Our system integrates these two methodologies in a manner that aims to explore their collaborative potential. Whereas conventional automatic code generation methods often operate in isolation, our approach integrates them with advanced business process modeling techniques, creating a more holistic development environment. The “Process Design Editor” is a key component of our system, designed to offer enhanced capabilities beyond traditional tools. The drag-and-drop methodology of the “Process Design Editor” facilitates faster visual design and offers a different approach for developers and business analysts in conceptualizing business processes. The ability to visualize without coding is not just a convenience—it is a paradigm shift. By merging these domains, our system promises speed, flexibility, and consistency and offers a methodology greater than the sum of its parts. In this research, we build upon existing software engineering methods and introduce a new methodology that offers a different perspective on development approaches.

This study introduces a novel system that integrates automatic code generation with process management for web-based applications. At its core, the system employs an intricate use of stored procedure patterns, plug-ins, HTML templates, and a template library. These elements work in concert to produce a polished final product. Open-source components, crafted in a standardized format, form the backbone of this initiative. They ensure that the resultant application is customizable and aligns with industry benchmarks. This process leans heavily on a well-defined database structure, instrumental in driving the automatic generation of CRUD operations, a mainstay in most web applications. The embrace of open-source modules and a coherent database structure accentuates this approach. There is a pronounced focus on industry-standard methodologies such as CRUD operations and JSON-based data modeling. The discussion explores the intricacies of designing and executing workflow processes tailored for web platforms. The "Process Design Editor," an intuitive web application interface, is central to this narrative. Drag-and-drop capabilities provide developers with a platform to craft intricate process visual flows. Illustrative guides in the form of interactive flowchart diagrams elucidate the relationships between various process tasks. From conditional flows activated by specific triggers to singular and multi-task sequences, these diagrams capture the flexibility of workflows, accommodating both straightforward and multifaceted operational needs. Moreover, this discourse delves into the tools that actualize these workflows, the technical intricacies of translating these visual flows into mxGraph models, and the pivotal role of robust databases and mapping tables. Including RabbitMQ and the SAGA architectural model further guarantees system efficiency and responsiveness. In light of these innovations, it is imperative to note that automated code generation, albeit potent, is not the panacea. Business Process Modelling Notation (BPMN) is pivotal for modeling business processes and ensuring the application's longevity. Our research tackles a BPMN-centric web application development approach rooted in automated code generation [1]. The platform we introduce aims to vividly present business processes to stakeholders. Notably, our drag-and-drop methodology facilitates application development, even for those sans programming expertise. The consistency in our database design further refines data integration processes. Standardization in the database optimizes data integration processes. The system enables rapid modeling of business processes, saves costs, and accelerates the development process. A major thrust of our approach relies on open-source components and a cogently structured database, emphasizing industry-standard methodologies such as CRUD operations and JSON-based data modeling. Our vision, supported by preliminary findings, highlights a staggering 98.68% improvement in developmental processes, a 95.84% enhancement in test conditions, and a 36.01% reduction in code size. Successful test scenarios confirm the effectiveness of this system. The study includes a literature review, system introduction, test cases, and results, aiming to shed light on the potent amalgamation of automated code generation and business process modeling, setting the stage for subsequent sections of this research paper.

## 2. Related Work

In the last five years, many studies on automatic code generation have been carried out, and code generation has been successfully applied in different fields. There are many critical works in the literature on automatic code generation, business process management, web application modeling, UML modeling, and database modeling. This section reviews some critical works that have significantly contributed to these areas. Paolone et al. [2] propose a methodology that simplifies web application modeling and design by adopting the Model-Driven Architecture (MDA) approach. This approach aligns with our focus on integrating automated code generation and business process management. Durai et al. [3] have developed a specific tool for converting UML models into XML Schema Definition (XSD) format, which indicates the link between data modeling and code generation. Yongchareon et al. [4] provide an in-depth analysis of the relationship between user interfaces and business processes, highlighting the potential synergy between automated code generation and business process management. In this context, Idrees et al. [5] consider the potential

advantages of a visual programming language when adapting the automatic code generation process to different programming languages. The BPMN modeling approach has been adopted by Mythily et al. in [6] and Zafar et al. in [7]; both groups studied the integration of code generation within business processes. Núñez et al. in [8], based on the Model-Driven Development (MDD) approach, elaborate on the transition from platform-independent to platform-specific models. Sunitha et al. [9] and Simon et al. [10] investigated the effects of UML-based modeling on code generation, while Apostol et al. [11] focused on the costs and challenges of UML-derived code. In this framework, Sánchez-Morales et al. [12], Chaber et al. [13], and Burak et al. [14] address automated code generation processes and demonstrate the potential for integration with different platforms and paradigms. Bocciarelli et al. in [15] explored the relationship between automatic code generation and database modeling, while Dinkelbach et al. in [16] discussed optimizing code generation processes using template engines. Sebastián et al. [17] and Ding et al. [18] have enriched automated code generation processes by adopting template- and transformation-based approaches. Hu et al. [19] discussed the impact of platform changes on code generation and how to deal with these changes. Yang et al. in [20] presented open code generation techniques with a template-based approach, showing how templates can be effectively used in automated code generation processes. Anuar et al. [21] detailed the relationship between automated code generation and web application development based on the MVC paradigm. Sunitha et al. in [22] examined code generation from UML activity diagrams, and this work provided important insights into how UML-based modeling can be considered in the context of code generation. This work has contributed significantly to a better understanding of the relationship between automated code generation, business process management, and modeling techniques and forms the basis for this research.

### 3. Architecture

This study proposes a system that covers both automatic code generation and process management for web-based applications. The system includes the automatic generation of web form pages, the definition of interactive flowcharts between the pages, and the generation of the required code. The method proceeds in two main stages. In the first stage (Section 3.1), tasks in specific formats are generated according to the set priorities. The second stage (Section 3.2) presents a structure in which automatically generated web pages are modeled in an interconnected process management framework. Figure 2 depicts the system architecture for web-based process management with automatic code generation.

#### 3.1. Automatic Code Generation to Create a Task

In order to implement this process effectively and efficiently, components such as stored procedure patterns, JS functions for plug-ins, HTML patterns, JSON schema, and the T4 Template library must be readily available. In addition, these components include libraries, templates, plug-ins, and other HTML view patterns. These components are open-source code written in a standardized way that conforms to standard form elements.

The functionality of the patterns encompasses components such as Plugin JS, HTML templates, stored procedures, and JSON schema. Plugin JS allows users to interact with JavaScript code that manages certain web features. HTML templates define the visual presentation of web page elements. JSON schema is a data structure that determines how data are stored and processed. These components can make the software development process faster and easier. They are also customizable and can be adjusted by developers to suit the project's specifics. HTML plugins and plug-ins are available, including ready-made JavaScript functions for each HTML element. Some examples of plugins are given in Table 1.



**Table 1.** HTML and JavaScript pattern templates.

Element	JavaScript Plugin	HTML Plugin
Element	JavaScript Plugin	HTML Plugin
Button	button.js	button.html
DatePicker	datepicker.js	datepicker.html
Form	form.js	form.html
Image Viewer	image_viewer.js	image_viewer.html
Selectbox	selectbox.js	selectbox.html
Input	input.js	input.html

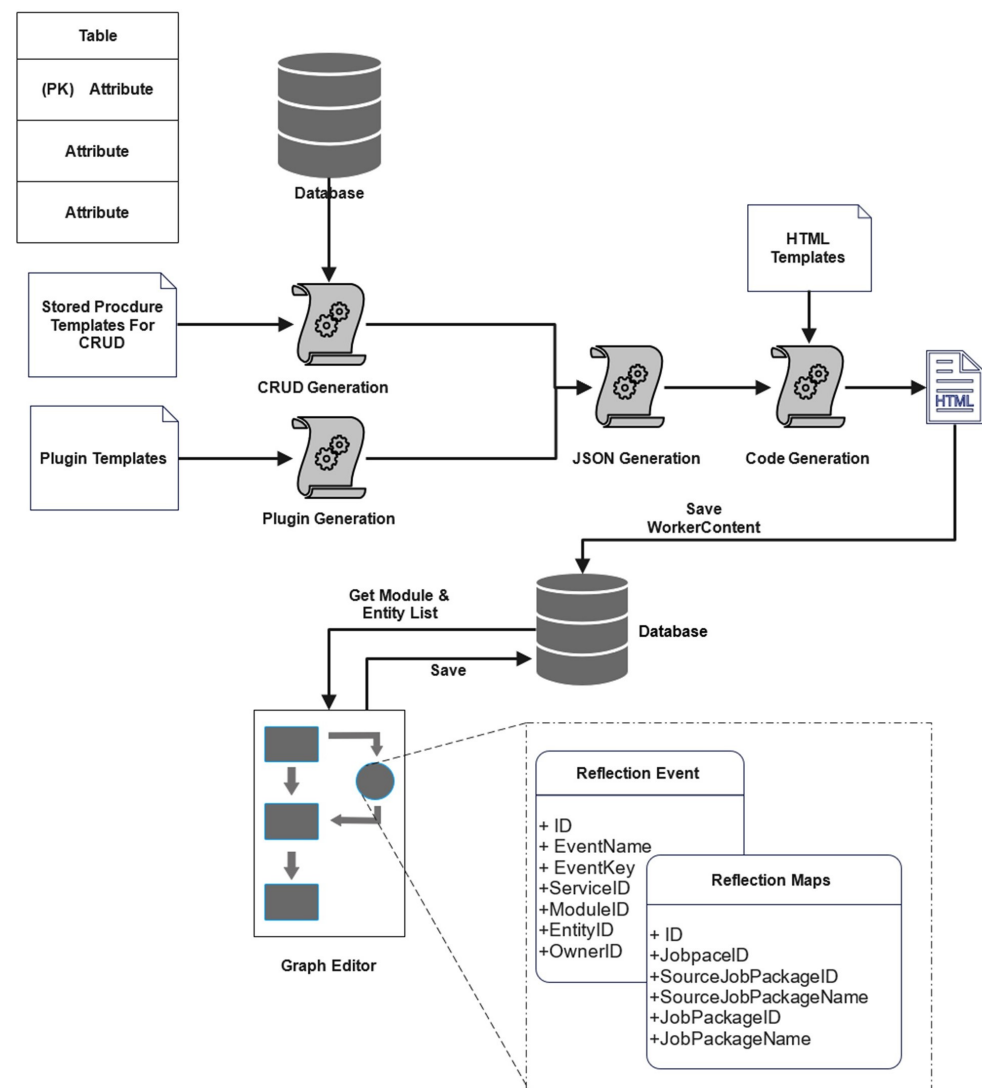
**Figure 2.** Stages of automatic code generation for system architectures.

Figure 3 shows the database schema used for automatic code generation. This database structure defines the main tables of Module, Plugin, Element, Function, Action, and Entity. Establishing relationships between these main tables is undertaken by tables defined with the suffix “Map”, for example, PluginFunctionMap and FunctionElementMap. However, tables with the word “type” in their name represent properties or values associated with the respective parent tables. In database management systems, the automatic generation of CRUD (Create, Read, Update, Delete) operations that carry out essential data operations provides efficiency gains in software development. Using ready-made stored procedure

templates facilitates these automatic code-generation processes in this context. Ready-stored procedure templates are given in Figure 4.

When the automatic code generation infrastructure is prepared, the first step is to add database tables for code generation. The table shown in Figure 5 has “Column Name, Data Type, and Allow Empty Values” columns. For example, in the “Leave” table, leave start date, leave end date, and description parameters are defined under “Column Name”. The relevant data types for these parameters are specified in the “Data Type” column. The “Allow Empty Values” column indicates whether specific fields are mandatory.

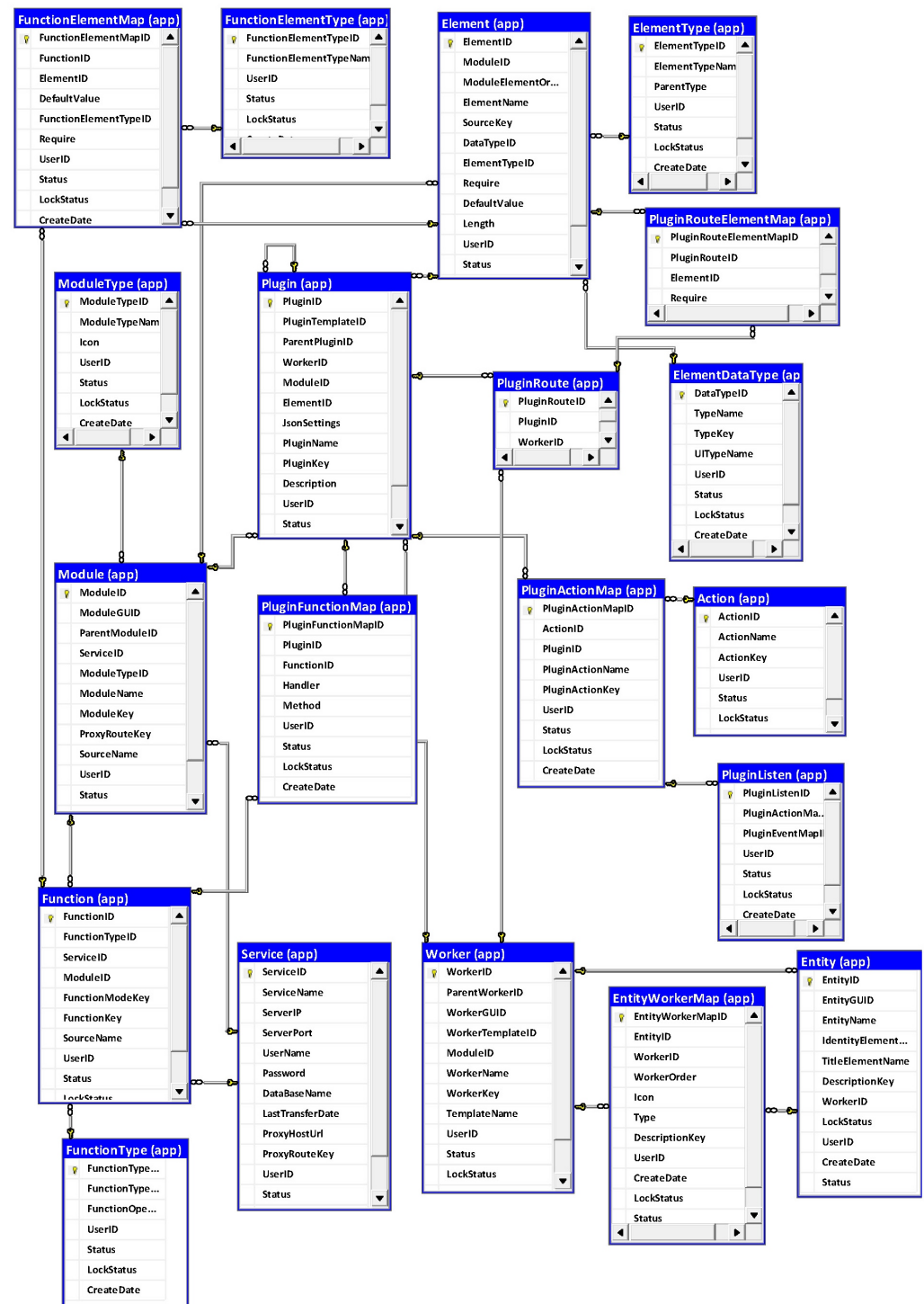


Figure 3. Database class diagram.

In the second stage, the automatic code generation process starts when the developer creates a data table. Based on the table name, the program automatically executes standard stored procedures. In this process, the development efficiency is increased by using predefined stored procedures. The pseudo-code below defines a connection string based on a specified service name, initializes database connections, and classifies tables according to specific criteria. This code generates SQL procedures for CRUD (Create, Read, Update, Delete) operations and specific procedures for specific tables. These procedures are automatically stored or updated in the database.

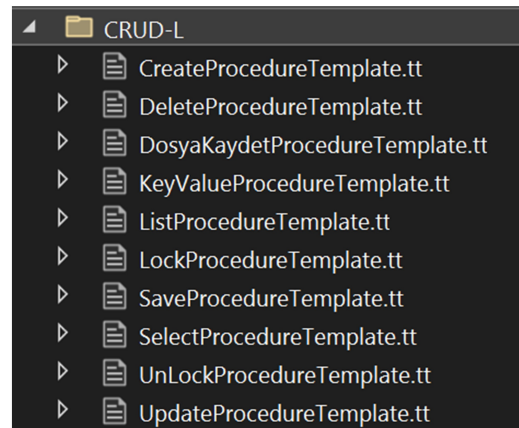


Figure 4. Stored Procedure Patterns.

	Column Name	Data Type	Allow Nulls
✖	StudentID	int	<input type="checkbox"/>
	StudentName	nvarchar(MAX)	<input type="checkbox"/>
	LockStatus	bit	<input checked="" type="checkbox"/>
	Status	tinyint	<input checked="" type="checkbox"/>
	CreateDate	datetime	<input checked="" type="checkbox"/>
	CreateUserID	uniqueidentifier	<input checked="" type="checkbox"/>

Figure 5. Manually created database table attribute list.

In the framework of this work, a stored procedure pattern specific to the Create operation is presented in Algorithm 1. This function creates a stored procedure that performs a Create operation for a given table in the database. If the specified procedure already exists, it is removed and redefined. SQL parameters are set based on the table columns, and a record is inserted during creation. The function returns the full text of the created procedure.

The function described in Algorithm 2 is used to manipulate columns in database tables and plays a central role in all stored procedures.

Each table column identifies unique columns, which are determined with the help of function, id, and meta, and then generates SQL parameter sequences based on this information. The generated parameters are combined according to a specified delimiter and returned as a result. The following function creates the specified database table's "SQL INSERT INTO" procedure. The function defines the required column names and parameters for each table column, excluding unique columns and key columns specified with the help of meta. Based on this information, the function creates the "INSERT INTO" and "VALUES" expressions, concatenates them according to a specified delimiter, and returns them as results. The generation of the stored procedure "Create" shown in the example above follows a similar approach for other stored procedures such as "Update",

“Read”, “Select”, and “Delete”. It takes table fields from the database according to a specific pattern and generates the stored procedure according to this pattern. Thanks to the “EXEC” command it contains, it saves the generated stored procedure directly to the database. The table in Figure 6 presents the details of the stored procedures generated automatically. “SPName” specifies the name of the stored procedure. One of the most critical sections, “SPGenerateScript”, contains the SQL code of the generated stored procedure. Finally, “SpTypeName” defines the type of operation the procedure performs.

---

**Algorithm 1:** Stored procedure generation algorithm
 

---

**Result:** SQL procedures Created, Updated, Read, Delete, Save in the database  
 Initial start-up and adjustments;  
 Setting ConnectionString according to ServiceName;  
 Initializing Tool and Service DB connections (tcn, cn);  
**if NOT isDatabaseReady then**  
     RETURN;  
**else**  
     Refresh the database object;  
     Initialize pointer variables;  
     Filter CRUDTables, PoolTables, etc. tables by criteria;  
     **if generate\_CRUD then**  
         Set all CRUD flags (generate\_C, generate\_R, etc.) to TRUE;  
         **for Every table in CRUDTables do**  
             **if generate\_C then**  
                 Initialize CREATEProcedureTemplate for table;  
                 Create the SQL script for the creation;  
                 Save or update the SQL script to the database;  
             **end**  
             **if generate\_R then**  
                 Initialize SELECTProcedureTemplate for table;  
                 Create SQL script to read;  
                 Save or update SQL script to database;  
             **end**  
             Something else can be done here (similar operations can continue for Update, Delete, Save)  
         **end**  
     **end** similar loops may continue for other flags such as generate\_KEY\_VALUE\_LIST, generate\_LOCK, etc.  
**end**

---

Results		Messages				
	SPGenerateID	SPName	SPGenerateScript	SpTypeName	SchemaName	TableName
1	7534	SP\$Student_CREATE	-----...	CREATE	app	Student
2	7535	SP\$Student_SELECT	-----...	SELECT	app	Student
3	7536	SP\$Student_UPDATE	-----...	UPDATE	app	Student
4	7537	SP\$Student_ONLYVALUE_UPDATE	-----...	ONLYVALUE_UPDATE	app	Student
5	7538	SP\$Student_SAVE	-----...	SAVE	app	Student
6	7539	SP\$Student_DELETE	-----...	DELETE	app	Student
7	7540	SP\$Student_LIST	-----...	LIST	app	Student
8	7541	SP\$Student_KEYVALUELIST	-----...	KEYVALUELIST	app	Student
9	7542	SP\$Student_LOCK	-----...	LOCK	app	Student
10	7543	SP\$Student_UNLOCK	-----...	UNLOCK	app	Student

**Figure 6.** List of generated stored procedures.

The stored procedures presented in Figure 7 are automatically added to the database by triggering the “SPGenerateScript” codes in the SPGenerate table. This mechanism provides fast and automatic preparation of the procedures required for CRUD operations.

```

+ app.SP$Student_CREATE
+ app.SP$Student_DELETE
+ app.SP$Student_KEYVALUELIST
+ app.SP$Student_LIST
+ app.SP$Student_LOCK
+ app.SP$Student_ONLYVALUE_UPDATE
+ app.SP$Student_SAVE
+ app.SP$Student_SELECT
+ app.SP$Student_UNLOCK
+ app.SP$Student_UPDATE

```

**Figure 7.** Stored procedures saved in the database.

---

#### Algorithm 2: TransformText

---

**Result:** GenerationEnvironment.ToString()

**Initialization:** Create a list named “columns” that includes unspecified columns in the table;

**Define** texts as a string list;

**Define** columnName as an empty string variable;

**if** “[table schema].[SP\$[table name]CREATE]” object exists **then**

    Create a new procedure delete command;

**else**

    Execute “EXEC [table schema].[SP\$[table name]CREATE]”;

**end**

**foreach** column in columns **do**

**if** column is Identity **then**

**if** identityWrite **then**

            Add “[column.Name]=NULL” to texties list;

**else**

**if** column name matches MetaHelp.CreateUserColumnName **then**

                Set columnName to MetaHelp.CreateUserParameter;

**else**

                Set columnName to “[column.Name]”;

                Add “[columnName]=NULL” to texties list;

**end**

**end**

**end**

**end**

**foreach** column in columns **do**

**if** column is not primary key **then**

        Add “[column.Name]” to insertSegment list;

**end**

**if** column name matches MetaHelp.CreateUserColumnName **then**

        Set columnName to MetaHelp.CreateUserParameter;

**else**

        Set columnName to “[column.Name]”;

        Add “[columnName]” to paramsSegment list;

**end**

**end**

Add “INSERT INTO [table.Schema].[table.Name](joined insertSegment)” to result list;

Add “VALUES(joined paramsSegment)” to result list;

Print joined result list using perLine delimiter;

---



The third step is to save the table created in the database via the Web interface. This process extracts the data, and the appropriate plugins are automatically created. These plugins define the design and functionality of HTML components (buttons, text boxes, etc.) in the user interface. In Figure 8, database tables are represented in a low-code platform. The rows marked in red color indicate that they have not been registered. The “Register” button at the end of each row allows the relevant record to be made.

15 StudentServices


Database			Module				
	Schema	Table	Mod...	Module Name	Module Type	Wr...	
23	app	Test	4774	Test	Normal	4	
24	app	Student	0				 Register
25	app	ThesisWorkRequest	4770	ThesisWorkRequest	Normal	4	
30	app	WeeklyWork	2674	WeeklyWork	Normal	10	

Figure 8. List of database tables in the Low-Code Platform.

During this process, each element’s plugins, actions, and functions are produced separately, and these productions are stored in the database. In the “Element” section, the essential attributes of each column in the database table are determined in detail. Among these attributes, “TypeName”, which defines the column’s data type, and “UITypeName”, which represents the equivalent of this type in the user interface, are particularly prominent.

Based on the ElementDataType table in Figure 9, default values for a given data type are stored in the Element table (Figure 10). The purpose of this approach is to systematically save the corresponding data type and default values for each element in the interface in order to be able to generate the corresponding data type for each element.

DataTypeID	TypeName	TypeKey	UITypeName
4	bit	bit	boolean
5	int	int	int
6	bigint	bigint	number
7	tinyint	tinyint	number
8	nvarchar	nvarchar	string
9	nchar	nchar	string
10	smallint	smallint	number
11	datetime	datetime	string
12	decimal	decimal	number
13	uniqueident...	uniqueid...	string
14	Nullable<...	Nullabl...	number
15	Int32	Int32	number

Figure 9. DataType data table.

ElementID	ModuleID	ElementName	SourceKey	DataTypeID	ElementTypeID	Require	DefaultValue	Length
15327	4773	LockStatus	LockStatus	4	9	0	((0))	1
15328	4773	Status	Status	7	8	0	((1))	1
15329	4773	CreateDate	CreateDate	11	15	0	(getdate())	8
15330	4773	StudentName	StudentName	8	7	1	NULL	-1
15331	4773	CreateUserID	CreateUserID	20	6	0	NULL	16
15332	4773	StudentID	StudentID	5	3	1	NULL	4

Figure 10. Element data table.

In Figure 11, the “Function” section stores the stored procedure names associated with the corresponding database table. The primary purpose of this section is to provide a list of which stored procedures will be triggered as a result of actions performed in the interface. For predefined operations such as Create, Update, and Delete, the relevant stored procedure names are stored in the database in this section.

	FunctionID	FunctionTypeID	ServiceID	ModuleID	SourceName
4651	8483	1	18	4773	SP\$Student_CREATE
4652	8484	4	18	4773	SP\$Student_SELECT
4653	8485	2	18	4773	SP\$Student_UPDATE
4654	8486	60	18	4773	SP\$Student_ONLYVALUE...
4655	8487	23	18	4773	SP\$Student_SAVE
4656	8488	3	18	4773	SP\$Student_DELETE
4657	8489	10	18	4773	SP\$Student_LIST
4658	8490	5	18	4773	SP\$Student_KEYVALUELI...
4659	8491	8	18	4773	SP\$Student_LOCK
4660	8492	9	18	4773	SP\$Student_UNLOCK

Figure 11. Function data table.

The “Plugin” section in Figure 12 offers the ability to create plugins for each element in the interface. It supports the creation of forms and the integration of buttons with functions such as “save” and “delete” automatically. Plugins are registered to specific ID values by associating them with the “PluginElementID” value. For example, when the data type in the data table “StudentName” is nvarchar, the appropriate HTML equivalent for this type is defined as “input” in the table “PluginTemplate” (Figure 13). Based on this information, an automatic plugin record is created. Plugins are prepared based on the predefined “Element” table and then designed for integration with the HTML content in the interface.

PluginID	PluginTemplateID	ParentPluginID	WorkerID	ModuleID	PluginName	PluginKey
38154	31	38147	8977	4773	StudentID identity input	StudentID_input
38155	1	NULL	8978	4773	Student form	Student_form
38156	17	38155	8978	4773	Student save button	Student_save
38157	18	38155	8978	4773	Student lock button	Student_lck
38158	19	38155	8978	4773	Student clear button	Student_clear
38159	21	38155	8978	4773	Student delete button	Student_delete
38160	26	38155	8978	4773	Student unlock button	Student_unlock
38161	29	38155	8978	4773	StudentName text input	StudentName_input
38162	31	38155	8978	4773	StudentID identity input	StudentID_input

Figure 12. Plugin data table.

PluginTemplateID	PluginTemplateName	PluginTemplateKey	Pattern
26	unlock button	unlock	button
27	int input	input	input
28	string input	input	input
29	text input	input	input
30	hidden input	input	input

Figure 13. PluginTemplate data table.

The “PluginFunctionMap” table (Figure 14) performs a critical association task between the “Plugin” and “Function” database tables. This table specifies how each plugin is associated with a particular function, which ensures that actions in the interface are consistently integrated with functions in the backend. A specially designed interface handler

called “UICrt” is used for the “Create” operation. This handler is mapped to “FunctionId” to trigger a specific “Function”. Thanks to this mechanism, the records specifying which actions trigger a stored procedure are stored in the “PluginFunctionMap” table.

PluginFunctionMapID	PluginID	FunctionID	Handler	Method
17414	38155	8483	UICrt	Create
17415	38155	8484	UISlct	Read
17416	38155	8485	UIUpdt	Update
17417	38155	8488	UIDlt	Delete
17418	38155	8491	UILck	Lock
17419	38155	8492	UIUnLck	UnLock

**Figure 14.** PluginFunctionMap data table.

“Action” (Figure 15) represents the actions of each plugin in the interface. It includes operations or actions that occur as a result of user interactions.

PluginActionMapID	ActionID	PluginID	PluginActionName	PluginActionKey
21028	10	38147	Student form Submit	Student_form_submit
21029	11	38147	Student form Delete	Student_form_delete
21030	5	38147	Student form Clear	Student_form_clear
21031	8	38155	Student form Lock	Student_form_lock
21032	9	38155	Student form Unlock	Student_form_unlock
21033	10	38155	Student form Submit	Student_form_submit
21034	11	38155	Student form Delete	Student_form_delete
21035	5	38155	Student form Clear	Student_form_clear

**Figure 15.** PluginAction data table.

At the core of the automation process are sophisticated data tables that define production parameters together with business rules. These data tables specify the directions of code generation within the automation, which templates or structural elements are preferred, and what kind of functionality the generated code will realize. In this context, you can find examples of a few tables below.

- **DefaultModuleWorkerTemplate:** Defines the structure with which the module and the worker template are associated. This table provides strategic information about how the code structure of modules should be created.
- **DefaultPluginAction:** Defines the specific actions that plugins will perform. These actions provide critical information about the functionality of the plugin.
- **DefaultPluginFunctionType:** Specifies the function types that plugins will contain. This table provides information about what types of algorithmic functions plugins should have.
- **DefaultPluginListen:** Defines the events that plugins should listen for. This specifies what kind of reaction mechanism the plugin should have in an event-based architecture.
- **DefaultPluginTemplate:** Specifies the basic templates to be used for plugins. This defines which template to reference during code generation.
- The functionality and configuration of plugins are often organized around specific templates and rules. In this context, the PluginTemplate and Action tables are an excellent example of this type of configuration.

The PluginTemplate table (Figure 16) defines the basic templates of plugins. For example, a plugin with a PluginTemplateID value of 1 indicates a form-based structure. A form-based structure is typically used to receive user input and process data. Therefore, it is natural for a form plugin to require specific actions, such as submit, delete, lock, and unlock.

PluginTemplateID	PluginTemplateName	PluginTemplateKey	Pattern	PluginTemplateCategoryID
1	form	form	form	3
2	list	list	list	3
3	datepicker	datepicker	datepicker	2
4	daterangepicker	daterangepicker	daterangepicker	5
5	datetimepicker	datetimepicker	datetimepicker	2
6	icheck	icheck	icheck	2
7	checkboxlist	checkboxlist	checkboxlist	4

**Figure 16.** PluginTemplate data table.

The Action table (Figure 17) identifies these actions. Each action has a unique ActionID associated with the DefaultPluginAction table (Figure 18) to determine which plugin template supports which actions. For example, the form template with PluginTemplateID 1 supports actions with ActionIDs 8, 9, 10, and 11.

ActionID	ActionName	ActionKey
8	Lock	lock
9	Unlock	unlock
10	Submit	submit
11	Delete	delete
12	AddItem	addItem

**Figure 17.** Action data table.

DefaultPluginActionID	PluginTemplateID	ActionID
6	1	8
7	1	9
8	1	10
9	1	11
10	1	5

**Figure 18.** DefaultPluginAction data table.

They automatically select the HTML template that best suits the user's needs based on specific business rules. In particular, when creating a data table, this automation platform automatically selects a form-based HTML template to ensure consistent input data collection. When the data set needs to be listed or queried, another listing-oriented template is preferred to ensure the data are presented appropriately. Workflows defined in the process editor also benefit from this automation. When a conditional workflow is created, the platform automatically adapts an HTML template with functional buttons such as "approved" or "not approved". In short, these automated code generation platforms select the most appropriate HTML template based on business rules, thus speeding up and standardizing the software development process. Such a structure allows the software to be modular and customizable. It also provides a framework for determining which software components fulfill which functions. This allows the software to have consistency, ease of maintenance, and scalability. In the fourth stage, JSON data modeling was adopted, a popular approach in response to today's data storage and cross-platform data transfer needs. The JSON generation process, triggered from the web interface through the code generation editor, automatically converts the data in the database into JSON format. As a result of triggering the "Generate JSON" command through the code editor of the low-code platform, the information of essential components such as Element, Plugin, Function, and Action is automatically converted into JSON format. This conversion process is structured through

the main sections of the JSON schema, namely Description, Plugins, and JobSpaceAction. The detailed structure of the generated JSON schema is shown in Figure 19.



Figure 19. JSON Schema.

In the fifth stage, the JSON conversion performed through the web interface is integrated into the system. The HTML code generation process performs automatic code generation based on JSON data sets. This coding is based on tags that define how the documents interact and how the content (text and images) is positioned. For example, for a button component, the relevant records in the database are automatically mapped to the specified action functions and plug-ins. The HTML template example in Listing 1 illustrates a templating mechanism used in contemporary web applications for dynamic data integration. The corresponding HTML element is decorated with different data-\* attributes. The {{ }} constructs inside these attributes represent variable or function calls of the templating language.

Listing 1. HTML template example for the 'div' section of a form-based workflow.

```

1 <div class="worker jobspaceaction-form worker-right-side"
2 data-workertype="jobspaceaction-form"
3 data-workerconnections="{{toJSON WorkerConnections}}"
4 data-assetkey="{{this. JobSpaceAction.Entity.Assetkey}}"
5 data-jobspaceactionid="{{this. JobSpaceAction.JobSpaceActionID}}"
6 data-jobspaceactionkey="{{this. JobSpaceAction.JobSpaceActionkey}}"
7 data-workername="{{this. JobSpaceAction.JobSpaceActionName}}">
8 </div>

```



The specified HTML segment reflects a typical component of process flow templates. This particular segment is directly associated with the JSON data structure in the Root.JobSpaceAction scope. The “{{ }}” notation used facilitates dynamic data injection, whereby the relevant JSON data elements are automatically injected into the template and blended with the specific information of the process flow. As in Listing 2, these placeholders are replaced with the corresponding values during templating, resulting in flexible and dynamic content.

**Listing 2.** Generated HTML code for the “div” section of a form-based workflow.

```
1 <div class="worker jobspaceaction-form worker-right-side"
  data-workertype="jobspaceaction-form" data-workerconnections=""
  data-assetkey="StudentID" data-jobspaceactionid="69"
  data-jobspaceactionkey="13df19af-26cb-465d-a9f5
2 -be2b25ef1244" data-workername="test">
```

The specified code examples are parts of a templating mechanism used for dynamic web content generation. The first code segment (Listing 3) reflects the Handlebars.js templating language used to dynamically generate buttons for specific actions by browsing plugins based on a given set of criteria. In particular, this segment creates buttons for the “clear”, “delete”, and “save” types defined within a “form” plugin.

**Listing 3.** HTML template example for the “job-action-content” section of a form-based workflow.

```
1 <div class="job-action-content">
2   <div class="actions">
3     <div class="form-action">
4       {{#each Plugins}}
5         {{#if_eq this.Description.Name 'form'}}
6           {{#each this.Plugins}}
7             {{#if_eq this.Description.Type 'clear'}}
8               {{#button this}} {{/button}}
9             {{/if_eq}}
10            {{#if_eq this.Description.Type 'delete'}}
11              {{#button this}} {{/button}}
12            {{/if_eq}}
13            {{#if_eq this.Description.Type 'save'}}
14              {{#button this}} {{/button}}
15            {{/if_eq}}
16          {{/each}}
17        {{/if_eq}}
18      {{/each}}
19    </div>
20    <div class="job-action">
21      <a class="btn btn-green jobactionlink"
22        data-key="{{this.JobSpaceAction.JobSpaceActionkey}}">
23        <i class="fa fa-send-o"></i>
24        <span data-i18n="ui:jsallk
25          {{this.JobSpaceAction.JobSpaceActionID}}">
26          {{this.JobSpaceAction.JobSpaceActionName}} Send Request
27        </span>
28      </a>
29    </div>
```

The second code segment (Listing 4) extends this template. Here, static HTML buttons are displayed for specific actions, while the properties and functionality of the buttons are fed with JSON data used in the background. This dynamic data injection allows the quick integration of customized interactive elements into the user interface in a suitable way.

**Listing 4.** Generated HTML code for the “job-action-content” section of a form-based workflow.

```

1  <div class="job-action-content">
2    <div class="actions">
3      <div class="form-action">
4        <button type="button" class="btn btn default"
5          data-plugin="button" data-type="clear"
6          data-plugin-settings="{ 'Common': { 'color': 'btn-default', 'icon': 'fa-recycle' }, '
          Definition': {} }" data-plugin-listens="" data-plugin-module=" 'Student' " name="
          Student_clear">
7          <i class="fa fa-recycle"></i>
8          <span data-i18n="ui:clear">Clear</span>
9        </button>
10       <button type="button" class="btn btn default"
11         data-plugin="button" data-type="delete"
12         data-plugin-settings="{ 'Common': { 'color': 'btn-default' }, 'Definition': {} }"
13         data-plugin-listens="" data-plugin-module=" 'Student' " name="Student_delete">
14         <i class="fa "></i>
15         <span data-i18n="ui:delete">Delete</span>
16       </button>
17       <button type="button" class="btn btn default"
18         data-plugin="button" data-type="save"
19         data-plugin-settings="{ 'Common': { 'color': 'btn-default' }, 'Definition': {} }"
20         data-plugin-listens="" data-plugin-module=" 'Student' " name="Student_save">
21         <i class="fa "></i>
22         <span data-i18n="ui:save">Save</span>
23       </button>
24     </div>
25     <div class="job-action">
26       <a class="btn btn-green jobactionlink"
27         data-key="13df19af-26cb-465d-a9f5-be2b25ef1244">
28         <i class="fa fa-send-o"></i>
29         <span data-i18n="ui:jsallk69">Send Request
30       </span>
31     </a>
32   </div>
33 </div>

```

The first code fragment (Listing 5) represents a raw template containing some placeholders. These placeholders are denoted by { } structures and are intended to be filled with JSON data, usually retrieved from a server or generated on the client side.

**Listing 5.** HTML template example for the “entity-form” section of a form-based workflow.

```

1  <div class="entity-form-content form">
2    {{#each Plugins}}
3      {{#if_eq this.Description.Name 'form'}}
4        {{#form this}} {{/form}}
5      {{/if_eq}}
6    {{/each}}
7  </div>

```

When applied in the second code block (Listing 6), the templating process of the generated HTML generates the final HTML form of the dynamic content. In the second block, we see a completed student form. While the form collects information such as the student’s name, certain “handler” functions are used to process and store this information. These functions are defined in the form’s data-plugin-proxies attribute and are used for various operations such as creating, reading, updating, and deleting student information. In addition, the data-plugin-listens attribute is used to specify the functions that should be performed in response to specific events in the form (e.g., saving student information). Figure 20 reflects an HTML view representing the final result of the automated code generation process. This visualization is a typical example of what an automated web interface can look like. This result is essential for developers and engineers to evaluate the success of automated coding processes.

**Listing 6.** Generated HTML code for the “entity–form” section of a form-based workflow.

```

1  <div class="entity-form-content form">
2      <form action="javascript:void(0)"
3          data-plugin="form"
4          name="Student_form"
5          data-plugin-settings="{ 'Common': {}, 'Definition': {} }"
6          data-plugin-proxies=" [
7              {
8                  'Handler': 'UICrt',
9                  'Service': 'StudentService',
10                 'Module': 'Student',
11                 'Function': 'Create',
12                 'Inputs': [ 'StudentName', 'CreateUserID' ],
13                 'Require': []
14             },
15             {
16                 'Handler': 'UISlct',
17                 'Service': 'StudentService',
18                 'Module': 'Student',
19                 'Function': 'Read',
20                 'Inputs': [ 'StudentID' ],
21                 'Require': []
22             }, {
23                 'Handler': 'UIUpdt',
24                 'Service': 'StudentService',
25                 'Module': 'Student',
26                 'Function': 'Update',
27                 'Inputs': [ 'StudentID', 'StudentName', 'CreateUserID' ],
28                 'Require': [] },
29             {
30                 'Handler': 'UIDlt',
31                 'Service': 'StudentService',
32                 'Module': 'Student',
33                 'Function': 'Delete',
34                 'Inputs': [ 'StudentID', 'CreateUserID' ],
35                 'Require': [] },
36             {
37                 'Handler': 'UILck',
38                 'Service': 'StudentService',
39                 'Module': 'Student',
40                 'Function': 'Lock', 'Inputs': [ 'StudentID' ],
41                 'Require': []
42             },
43             {
44                 'Handler': 'UIUnLck',
45                 'Service': 'StudentService',
46                 'Module': 'Student',
47                 'Function': 'UnLock',
48                 'Inputs': [ 'StudentID' ],
49                 'Require': []
50             } ]"
51      data-plugin-description="{ 'Name': 'form', 'Type': 'form', '
52          Settings': { 'Common': {}, 'Definition': {} },
53          'Route': { 'ServiceName': 'StudentService', 'ModuleName': '
54              Student',
55          'WorkerName': 'studenttalepformu2_10_09_2023_www' } }"
56      data-plugin-listens=" [
57          {
58              'ActionFunction': 'lock',
59              'ListenPlugin': 'lck',
60              'ListenEvent': 'Student_lck_click'
61          },
62          {
63              'ActionFunction': 'unlock',
64              'ListenPlugin': 'unlock',
65              'ListenEvent': 'Student_unlock_click'
66          },
67          {
68              'ActionFunction': 'submit',

```

```

66         'ListenPlugin':'save',
67         'ListenEvent':'Student_save_click'
68     },
69     {
70         'ActionFunction':'delete',
71         'ListenPlugin':'delete',
72         'ListenEvent':'Student_delete_click'
73     },
74     {
75         'ActionFunction':'clear',
76         'ListenPlugin':'clear',
77         'ListenEvent':'Student_clear_click'
78     }
79     class="fixed-height form-horizontal form-bordered
        form-label-stripped">
80 <div class="form-body">
81     <div class="form-group form-md-line-input ">
82         <label for="StudentName" class="col-md-3 control-label"
            data-i18n="Student:StudentName">StudentName</label>
83         <div class="col-md-9">
84             <div class="input-icon">
85                 <textarea data-plugin="input" class="form-control
                    " id="StudentName" name="StudentName"
                    data-plugin-settings="{ 'Common':{ 'type': '
                        textarea', 'icon': 'fa-pencil-square-o ', '
                        icon_type': 'input'}, 'Definition':{}}"
                    data-plugin-listens=""></textarea>
86                 <div class="form-control-focus"></div> <span
                    class="help-block" data-i18n="
                        Student:StudentName_desc">
                        Student:StudentName_desc</span> <i class="fa
                        fa-pencil-square-o "></i>
87             </div>
88         </div>
89     </div>
90     <input data-plugin="input" id="StudentID" type="hidden"
        class="identity-class" name="StudentID"
        data-plugin-settings="{ 'Common':{ 'hidden': 'true ', 'Class
            ': 'identity-class'}, 'Definition':{}}"
        data-plugin-proxies="" data-plugin-listens="">
91 </div>
92 </form>
93 </div>

```

*i* form veri girişiniz için hazırlandı.

**Figure 20.** GeneratedHTML web page view.

This templating approach illuminates how the data and presentation layer are effectively separated and integrated with dynamic web applications. This methodology significantly facilitates code readability and maintainability while increasing the flexibility of the application. In addition, we observe that dynamic HTML generation is possible in just a few steps by incorporating a manually created database into the low-code platform with the “Register” command and then using the “Generate JSON” and “Generate HTML” commands. This dramatically increases the efficiency and speed of development processes. The integration between the interface and the backend is realized using the RESTful web service. This methodology unlocks the potential of automatically coding existing stored procedures based on database tables. The generated code is compatible with RESTful service protocols. Following the automated coding process, the resulting web service code is compiled, and the DLL file generated due to this compilation must be hosted on the target server. The automation process is presented in the pseudocode below. In this pseudocode, the “Create” and “Update” methods are examples of the web service code

pattern (Listing 7). We can observe that the CreateMethodName, MethodReturnType, table name, and model variables are dynamically assigned; these variables are populated based on the corresponding database table, thus enabling automatic code generation.

**Listing 7.** Web service code pattern.

```

1  [Route("<#=MetaHelp.CreateMethodName #>")]
2  [HttpPost]
3  public <#=MetaHelp.MethodReturnType#> <#=MetaHelp.CreateMethodName #>
4  (SP_<#=table.Name #>_SELECT_Result model)
5  {
6      return db.SP_<#=table.Name #>_CREATE(
7          <#=NETHelp.WriteCreateClause(table, "model") #>).FirstOrDefault();
8  }
9  [Route("<#=MetaHelp.UpdateMethodName #>")]
10 [HttpPut]
11 public <#=MetaHelp.MethodReturnType#> <#=MetaHelp.UpdateMethodName #>
12 (SP_<#=table.Name #>_SELECT_Result model)
13 {
14     return db.SP_<#=table.Name #>_UPDATE(
15         <#=NETHelp.WriteUpdateValues(table, "model") #>).FirstOrDefault();
16 }

```

The automatically generated code of the ASP.NET Web API is essential in data integration. However, it should be noted that this automatic process requires some manual steps. In particular, the DLL obtained after compilation must be manually integrated into the server. This is an indispensable step for efficient data transmission between the client and the backend.

One of the notable advantages of our implementation is the method we have adopted for data exchange. We have fragmented the delivery process rather than delivering all assets in a single monolithic chunk. HTML, CSS, and data are retrieved separately, ensuring swifter loading times. No matter the expansion of our site, each time a webpage is accessed, it necessitates packets of JavaScript, HTML, CSS, and other data. Our pages load with remarkable speed by fetching these assets individually from the server and database. Moreover, our system works with a Content Delivery Network (CDN) methodology. It verifies content versions and sends requests only for the changed assets. While this boosts performance, it is pivotal to recognize that scalability presents its unique challenges, which we tackle using distinct techniques.

In Figure 21, HTML and JSON codes in the Source field in the WorkerContent table are integrated into these low-code platforms to support the dynamic content creation and display process. Regarding the functioning, firstly, code fragments in HTML and JSON format are saved in the specified database. These codes contain the structures and features that the user will use in their applications on the web. These saved code fragments are pulled from the database on demand by the platform and dynamically rendered on the user's interface. This process enables rapid prototyping, testing, and final release of the application. At the same time, centralized storage of code in a database brings operational advantages such as version control, backup, and deployment. As a result, this integration capability of low-code platforms accelerates application development processes, minimizes errors, and provides users with a more flexible development environment.

WorkerContentID	ContentHandlerKey	WorkerID	ContentTypeID	Source	Version
167393	studentalepformu2_10_09_2023_www	8978	3	{ "Description": { "Name": "studentalepfor...	4
167394	studentalepformu2_10_09_2023_www	8978	3	{ "Description": { "Name": "studentalepfor...	5
167395	studentalepformu2_10_09_2023_www	8978	1	<div class="worker jobspaceaction-form worker-r...	2

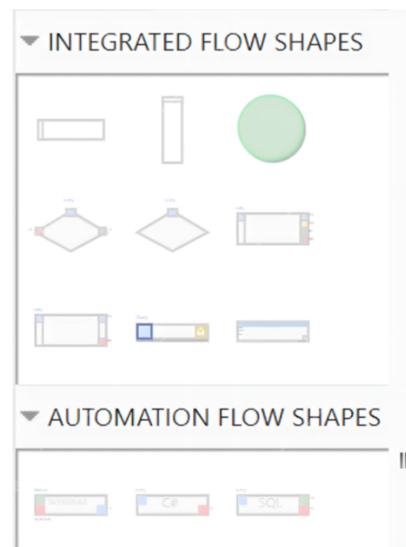
**Figure 21.** WorkerContent data table.

### 3.2. Creating a Workflow Process in a Web Environment

In the implementation of the process, firstly, the tasks are detailed. Then, using the Web Application interface called "Process Design Editor", the relationship and interaction flow

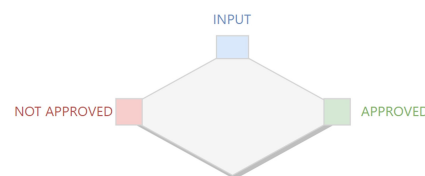


between these tasks are designed. This editor supports drag and drop, providing developers with an essential convenience in dynamically visualizing and editing the process. In Figure 22, taken as a reference, we can examine how users interact with flowchart flow diagrams to design a business process. Each symbol and flowchart represents a specific functionality or part of a process.



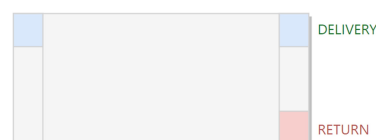
**Figure 22.** Flow diagram elements.

The conditional flow, shown in Figure 23, enables the workflow to follow different paths as a specific condition is fulfilled. This methodology supports dynamic routing in the workflow based on specific conditions. Binary decision mechanisms typically evaluate the conditions. This fulfills the requirements for branching and merging workflows so that processes become more flexible and allow the flow to be managed accurately.



**Figure 23.** Condition flow diagram element.

Single Task Flow, depicted in Figure 24, is a workflow model in which each task is sequenced based on the completion of the previous one. This model is ideal for simple, low-complexity processes where tasks build on each other and occur sequentially. In this structure, where each task has a specific priority, high efficiency is achieved with the correct sequencing.

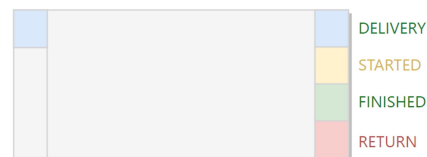


**Figure 24.** Single Task Flow elements in condition scenario.

The Multi-Task Flow presented in Figure 25 represents a workflow model in which multiple tasks are performed in parallel. This model emphasizes the independence of tasks within the process so that simultaneous progress is achieved without waiting for each task to complete. This approach is preferred for complex processes where waiting times must be

minimized. The creation of the workflow is realized using specific design tools. These tools help place the workflow steps sequentially with “drag and drop” features. Workflow steps represent specific actions, such as sending an e-mail or retrieving data from a database. This process includes the following steps:

- The purpose of the workflow is defined.
- Specific steps and their dependencies are identified.
- A flow diagram is created showing the connections between the steps.
- The inputs and outputs of the workflow are defined.
- The workflow is validated by testing.



**Figure 25.** Multiple Task Flow elements in condition scenario.

Order reception, processing, dispatch, invoicing, and payment confirmation. These steps proceed with specific inputs and are performed under predefined conditions. In Figure 26, the process editor has a panel that defines the characteristics of each flow, the associated service module, and the associated data package. The “Task Form” contains information retrieved from the Service and Module databases, while the “Task Actions” correspond to actions in the Flowchart.

**Figure 26.** Characteristics of a workflow.

Using the same example, the diagram in Figure 27, the process screen shows the workflow process. In order to make a design, the design is realized by dragging and dropping the elements in the panel in the design editor. An example of a “Leave Request Form” is given in Figure 27. This means that students are associated with the Leave Request Form module, and the data for each workflow are retrieved from the “Student” database.

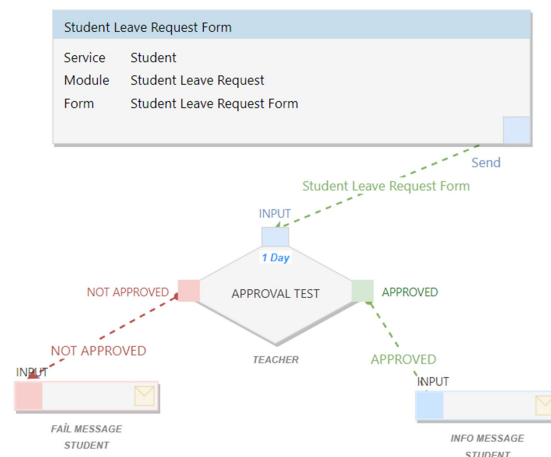


Figure 27. Work area of the Process Design Editor.

Definitions are made for manually added elements in the panel in the workspace. The completed design is converted to the mxGraph model. mxGraph is a JavaScript library for graphical interface designs and diagrams, enabling the creation of web-based visual models. It is known for its customization and integration capabilities. Listing 8 presents an example converted from workflow to mxGraph model; the “Confirmed” action indicates the next steps.

The coded version of the workflow model shown in Figure 28 is given in Listing 8. Each workflow element is identified with a unique identification number. The descriptions of the fields in the code example are as follows:

Listing 8. The mxGraph model of a workflow given in Figure 28.

```

1  <mxCell
2    id="0353db73-9856-4f8b-a4a3-a370c37d1502"
3    value="APPROVED"
4    <mxGeometry x="1"y="0.5"width="23"height="23"relative="1"
5      as="geometry">
6        <mxPoint x="-20"y="-12"as="offset"/>
7    </mxGeometry>
8    <JobConnectionDto as="data">
9      {
10        "obj":{
11          "AssetKey":"JobActionID",
12          "AssetValue":1767,
13          "Connect":"out",
14          "JobSpaceID":230,
15          "EventTypeID":2,
16          "Title":"Approved",
17          "TitleFull":"testonay",
18          "StyleFormater":"accept",
19          "_:{
20            "JobActionID":1767,
21            "JobPackageID":5521,
22            "JobActionName":"Approved",
23            "JobActionKey":"0353db73-9856-4f8b-a4a3-a370c37d1502",
24            "IsBaseAction":true,
25            "UserID":"4f7ae81d-96ce-4217-bfef-fd687328ca79",
26            "JobPackageName":"testonay",
27            "Durum":1,
28            "LogicValue":1,
29            "SortOrder":99
30          }
31        }
32      }
33    </JobConnectionDto>
34  </mxCell>

```

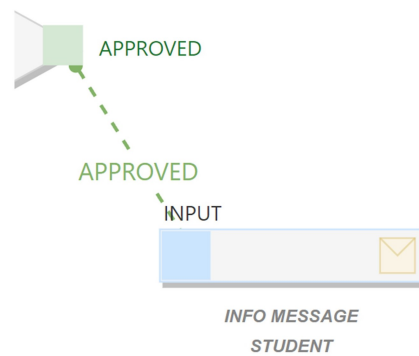


Figure 28. A sample workflow created using workspace.

Here are some of the properties for workflow items:

- ID: The unique identification number of the items.
- Value: The name of the item, such as “Approved”.
- AssetKey: The unique identifier of the asset.
- JobSpaceID: The ID of the workspace.
- EventTypeID: Defined type of the event.
- JobActionID: ID of the action.
- JobPackageID: Number of the data package to use.

When the user operates, the relevant information is retrieved from the database in real time. With this information, workflow processes are created using the graphical design tool. Finally, the created model is transformed with mxGraph and saved in the database (Figure 29).

GraphEditorSourceID	Name	Source
41	Consultant Teacher Request P...	<mxGraphModel dx="1113" dy="658" grid="1" gridSize="10" guides="1...
42	Student Permission Request Pr...	<mxGraphModel dx="2237" dy="1322" grid="1" gridSize="10" guides="...
43	Double Major Request Process	<mxGraphModel dx="1595" dy="937" grid="1" gridSize="10" guides="1...
44	Course Drop Request Process	<mxGraphModel dx="1963" dy="679" grid="1" gridSize="10" guides="1...
45	Repair Request Process	<mxGraphModel dx="934" dy="552" grid="1" gridSize="10" guides="1" ...
46	Work Request Process	<mxGraphModel dx="1382" dy="817" grid="1" gridSize="10" guides="1...
47	Student Leave Request Approv...	<mxGraphModel dx="1909" dy="1140" grid="1" gridSize="10" guides="...

Figure 29. GraphEditor source data table.

Each flowchart is saved in the database after conversion into a Business Model. The Business Model is essential for automating, coordinating, and scheduling business processes. This model specifies the steps required by the work, the sequence of these steps, the relevant data, and the actions to be performed at each step. This automation allows operations to be performed quickly, efficiently, and without errors. The transformed Business Model is stored and labeled in a separate database. This labeling is critical in defining and guiding complex processes with multiple actions. The automatically generated HTML element contains a fundamental property called “data-jobspaceactionkey”. This key is directly related to the ReflectionEventKey in the database and is called JobActionKey. ReflectionEventID information is retrieved from the database through this specific key when any action is triggered in the user interface. The resulting ID is used with the EntityID to guide further process steps (Listing 9).

Listing 9. Generated HTML code with the job action key.

```

1 <div class="worker jobspaceaction-form worker-right-side"
  data-workertype="jobspaceaction-form" data-workerconnections=""
  data-assetkey="StudentID" data-jobspaceactionid="68"
  data-jobspaceactionkey="111a9f99-ecea-42e3-8e2d-51f8ae1e8a67"
  data-workername="Student Request Form">

```

Within the structure in which the processes are defined, the relational data expressing the transitions from one process to another are already stored in the database and are retrieved from this database. This relational data set is stored in the ReflectionEvent table (Figure 30). In the example, jobspaceactionkey on the HTML side and ReflectionEventKey in the data table are the same.

ReflectionEventID	ReflectionEventName	ReflectionEventKey	ServiceID	ModuleID	EntityID	OwnerID	OwnerGUID
3235	Öğrenci İzin Talep Formu > ...	111A9F99-ECEA-42E3-8E2D-51F8AE1E8A...	46	2669	5122	68	111A9F99...
3236	Öğretmen İzin Onay Formu ...	0353DB73-9856-4F8B-A4A3-A370C37D1502	46	2669	5053	1797	90A87156...
3237	Öğretmen İzin Onay Formu ...	4B290B96-B12C-4825-8E69-B0133769CB46	46	2669	5053	1798	90A87156...
3238	Müdür İzin Onay Formu > O...	0353DB73-9856-4F8B-A4A3-A370C37D1502	46	2669	5053	1799	B6642703...
3239	Müdür İzin Onay Formu > O...	4B290B96-B12C-4825-8E69-B0133769CB46	46	2669	5053	1800	B6642703...
3240	test > Kayıt	13DF19AF-26CB-465D-A9F5-BE2B25EF1244	46	3669	6053	69	13DF19A...
3241	Permission Approval Form ...	0353DB73-9856-4F8B-A4A3-A370C37D1502	46	3669	6051	1801	D5B6A58...
3242	Permission Approval Form ...	4B290B96-B12C-4825-8E69-B0133769CB46	46	3669	6051	1802	D5B6A58...
3243	School principal Permission ...	0353DB73-9856-4F8B-A4A3-A370C37D1502	46	3669	6051	1803	13E9C50...

Figure 30. ReflectionEvent data table.

When switching to the next screen, the required screen information is retrieved through a specific ReflectionEventKey and presented to the user (Figure 31).

ReflectionMapID	JobSpaceName	SourceJobPackageName	SourceJobPackageID	JobPackageID	JobPackageName
6066	Student Talep Bildirim alanı	Permission Approval Form	5537	5538	School principal Permission Approval

Figure 31. ReflectionMap data table.

The “ReflectionMap” table is a critical mapping table for centralized management of the automation processes of information systems. The primary function of this table is to determine which process or screen will be activated according to the results of user actions.

- ReflectionMapID: Unique identifier of the record.
- JobSpaceName: The name of the process space.
- SourceJobPackageName: The name of the starting package from which the action is started.
- ReflectionEventName: Identifies the triggered event, usually indicating which phase the form is in.
- JobPackageName: Indicates which package will be activated after the triggered action.
- JobSpaceID: Unique ID of the process space.
- EntityID: Indicates the screen or process step to be activated.
- ReflectionEventID: The unique ID of the triggered event.

This table is designed to effectively manage the processes corresponding to user actions in the automation process. It determines which process or screen is activated when a user performs a specific action. It also provides a mapping between actions and results, which allows the automation processes within the system to proceed smoothly. This integrated structure allows the system to manage automation processes more efficiently while allowing the user a more fluid and seamless experience. User actions on the screen or form are instantly processed through the RabbitMQ messaging infrastructure. RabbitMQ is frequently used in microservice architectures with its flexible and secure protocols. For example, in an educational institution, the permission form filled out by the student is immediately notified to both the administration and the relevant teachers via RabbitMQ. This process occurs thanks to RabbitMQ’s publish/subscribe model, where messages are usually packaged in JSON or XML format (Listing 10).

Listing 10. RabbitMQ: publish/subscribe messaging with JSON and XML.

```

1  data = {
2      "Name": "John Smith",
3      "permission_type": "sickleave",
4      "start_date": "2023-04-18",
5      "end_date": "2023-04-25"
6  }
```



The third step is to send the data packet to a specific queue:

```
# Specify the queue name and the data packet
queue_name = 'permission_form' message = json.dumps(data)
# Send message to queue
channel.basic_publish(exchange="", routing_key=queue_name, body=message)
# Close the connection connection.close()
```

Messages are sent in JSON format to the permission\_form queue using the basic\_publish function and distributed to connected consumers. This communication is coordinated under the SAGA architecture and state machine. SAGA is an architectural model for microservice-based, autonomous, and decentralized applications. While each microservice runs on its database, the SAGA transaction execution manager ensures data consistency. This manager maintains data integrity when an error occurs by rolling back transactions. This architecture supports the independent development and scaling of microservices. Concurrent interaction is realized through the SignalR framework. The basic steps followed in the process of construction are as follows:

- **Data Modelling:** this is essential to design database structures. The database schema is visualized with the data modeling tools provided by the software used.
- **Process Definition:** tools such as MxGraph are used to design business processes graphically.
- **Interface Design:** the application's user interface is prepared with the visual interface design module provided by the software.
- **Determination of Business Rules:** rules regarding business logic are defined with the relevant software modules.
- **Testing and Execution:** the prepared process is tested to check its accuracy and functioning.
- These steps are carried out with software tools that enable processes to be built effectively.

#### 4. Experimental Method and Evaluation

In the first phase of the experiments, web application design was performed manually. Then, web application design was realized using automatic coding. The experiments' evaluation criteria were determined, and are presented in Table 2.

**Table 2.** Evaluation criteria and explanations.

Name	Description
Development Process Time	Time spent in web application development
Number of Test Runs	Number of test runs performed during implementation
Number of Errors	Number of errors made during implementation
Number of Code Lines	Number of lines of code (LOC) of the generated HTML file

Below are the items for manual and automated processes:

Manual Process Stages:

- **Database Design:** this stage is where the database foundations, which are at the heart of the software, are laid.
  - **Defining Tables:** the phase in which the structure in which the information to be stored in the database will be stored is determined.
  - **Creating Stored Procedures:** functions and commands prepared to perform database operations faster and more effectively.
- **Front End Development:** at this stage, the software's user interface is designed.
  - **Creating HTML:** the part where the basic visual structure of the website is created.
  - **CSS Integration:** the stage where the visual features of the website are stylized.

- JavaScript Coding: coding language used to interact with the user and add dynamic features.
- Server Side Development: server-related operations and optimizations are performed in this stage.
  - Receiving the Preliminary Workflow Data Package: receiving and processing the data packets of the first steps in the workflow.
  - Server Side Programming: writing the codes that process the user's requests and interact with the database.
  - Model Compatible Transformation and Packaging: packaging the data following the workflow model.
  - API Development: designing interfaces to optimize data exchange.
  - API File Compilation and Server Integration: integration of the prepared API into the server.
  - Stored Procedures and Database Operations: development of queries and procedures used to interact with the database.
  - Dynamic Screen Orientation Coding: coding allows the screen to be orientated according to user interaction.

#### Automated Process Stages:

- Table Definition: manual definition of database tables.
- Running Functions for CRUD: functions that automatically perform CRUD operations ("Create", "Read", "Update", and "Delete" stored procedures) are created automatically by clicking a button to run the function that allows you to create them quickly and error-free.
- Selecting and Saving a Data Table from the Low-Code Platform: After logging in to the low-code platform, users select the relevant data table in the database. This table is displayed with an automatically generated user interface (UI). After selecting the data table, users click a "register" button. This triggers database transactions.
- Running the Function to Automatically Generate API Codes: Automatic generation of API codes that enable data exchange between the server and the client. This automatic generation process is performed only once for a single data table. It is performed only once for several tasks in a given process.
- Opening the Process Editor on the Low-Code Platform: on the low-code platform, opening the process editor and selecting the desired workflow and adding it to the process editor.
- JSON and HTML Generation: Using the code editor available on the low-code platform for the selected workflow, users click the "Generate JSON" and then "Generate HTML" buttons. These steps automatically generate how the data will be displayed in JSON and HTML.
- Saving JSON and HTML Codes Generated from Low-Code Platform to the Database: storing the generated codes in the database.
- Establishing Workflow Connection in Process Editor: establishing the connection of workflows.

In the implementation of automated processes, the process can take approximately 2 min longer at the beginning of each new process due to the need to generate the API and put the generated compiled file (DLL) on the server. This additional time occurs in the first workflow due to the data import used in this process. However, there is no need for such a wait in the following workflows, because we use the same database table. According to the workflows defined in the process editor, the HTML configuration is also automatically shaped. For example, when you add a "form" workflow, the necessary input elements are automatically added with buttons such as "save", "delete", and "clear". However, special buttons such as "approved" or "not approved" are added for a conditional workflow. This automatic configuration enables the process to be highly optimized and ensures that transactions are carried out quickly and efficiently.

The metrics of the development process directly affect the effectiveness of the process. The metrics mentioned by Akbulut and Toprak [23] and Possatto and Lucrédio [24] have contributed significantly to the efficiency of this process. In particular, the bug count metric is directly linked to the developer's productivity, and the code size metric reflects the system's performance. At the same time, it is also known that increasing the code size increases the potential for errors. A practical scenario is as follows. This scenario was implemented with both manual and automated methods. The data collected during the implementation were analyzed according to the evaluation metrics. The analysis results show that the automatic code generation method can offer developers a faster and more effective process than the manual method. This advantage of automated methods increases the effectiveness of the process. The results of the experiment reveal the following essential findings in the comparison of automated and manual code development processes:

- **Development Process:** The average manual development time was 227.6 min, but the automated process reduced this to 3 min. This represents an efficiency increase of 98.68% in the development process.
- **Number of Test Runs:** Automated code and process generation requires only two test runs, while the manual process requires many more test runs. This corresponds to an improvement of 95.84%. In the manual method, this number is increased due to correctness testing of different stages, potential bug fixes, and repeated testing. The main advantages of automating the tests are shortening the development process and reducing the use of the central processor.
- **Number of Bugs:** When the error rates of the automated and manual code generation processes were compared, it was found that various errors occurred in the manual process, but the error rate in the automated process was zero. The detection of these errors is based on the sum of the user-related errors specified in the "error" section when run in the Visual Studio IDE environment. This indicates that the automated process is more reliable than the manual one. Furthermore, this finding implies that the number of errors is directly related to the number of test runs. A high defect rate is a factor that drives continuous test execution due to the need to fix defects.
- **Code Size:** The automatic code generation process was found to have a 36.01% more compact code size than the manual process. This is due to the efficient management of repetitive code blocks in the automated process. This optimization increases the speed of the process and prevents unnecessary code redundancy.

The findings show that automatic code generation offers more advantages than manual approaches. Based on these data, it is observed that automated code generation saves both time and cost in development processes. Furthermore, this automation approach significantly reduces error rates, with fewer iterations required in the testing process. From a computer engineering perspective, this is a critical method to increase efficiency and reliability in software development processes.

## 5. Conclusions and Comments

Automated code generation is receiving increasing attention in the software engineering discipline. Developers and researchers are adopting automated code-generation strategies to improve code quality and implementation efficiency. Especially in large-scale projects, manual code-writing processes can be time consuming and more prone to errors.

In our experiments, we compared the efficiency and accuracy of manual (Man) and automated (Auto) coding processes for various tasks, as illustrated in Table 3. Our results show that automated code generation significantly reduces development time; for instance, the Permission Request Process that took 960 min manually was reduced to just 13 min using the automated approach. The number of test runs required also reduced significantly with automated code generation, with the average number decreasing from 20.8 in manual processes to 2 in automated processes. The number of errors dropped to zero in all automated processes, whereas manual coding had an average of 14.9 errors across the tasks.

Finally, the number of code lines was also optimized with automation, with an average reduction from 176.75 lines in manual coding to 113.1 lines in automated coding.

However, the advantages of automated code generation are accompanied by specific challenges. For one, complex code blocks automatically generated by the system can pose challenges during maintenance and update processes. There is also the concern that automation, built around specific standards and templates, could limit customization. Furthermore, the intricacies and in-depth details of some automated code generation systems might overwhelm novice software engineers, necessitating additional investments in training and adaptation processes.

**Table 3.** Experiment table.

Process Name	Development Process Time (min)		Number of Test Runs (#)		Number of Errors (#)		Number of Code Lines (#)	
	Man	Auto	Man	Auto	Man	Auto	Man	Auto
1. Permission Request Process	960	13	82	8	59	0	751	429
1.1. Permission Request Form	522	6	40	2	32	0	422	218
1.2. Permission Approval Form	312	3	27	2	21	0	249	151
1.3. Permission Approval Notification Form	63	2	6	2	2	0	40	30
1.4. Permission Refusal Notification Form	63	2	6	2	2	0	40	30
2. Service Request Process	1772	23	168	16	120	0	1370	929
2.1. Service Request Form	495	5	37	2	34	0	358	186
2.2. Task Notification Form	70	2	11	2	6	0	54	36
2.3. Service Manager Job Assignment Form	347	4	35	2	22	0	241	182
2.4. Request Closure Form	241	3	24	2	19	0	206	161
2.5. Request Editing Form	211	2	16	2	11	0	211	153
2.6. Service Manager Request Closure Approval Form	294	3	33	2	24	0	208	148
2.7. Request Closure Approval Notification Form	55	2	7	2	3	0	45	33
2.8. Request Closure Rejection Notification Form	59	2	5	2	1	0	47	30
Average Values	227.6	3	20.8	2	14.9	0	176.75	113.1

No-code and low-code approaches take automated code generation to a more abstract level. These approaches include drag-and-drop editors and component-based design techniques. The proposed methodology synthesizes these approaches and provides a comprehensive platform for automated code generation. Extending this approach may bring technical challenges, like the need for deeper analyses in database management, security, and authentication. In light of our implementation strategy and the integration of CDN technology, the optimized data exchange method accelerates performance. It solidifies our system's scalability, showcasing the possibility of seamlessly blending speed with expansive growth in software systems. In summary, while automated code generation offers significant advantages regarding development time, test runs, error rates, and code optimization, it is crucial to be mindful of the challenges presented, especially when considering its broad adoption. Future research should focus on mitigating these challenges while refining and expanding the techniques to cater to industry needs.

In subsequent research, a pivotal focus will be on deploying and optimizing automated code generation techniques in enterprise-scale infrastructures, where numerous forms, complex workflows, and intricate integration processes necessitate advanced algorithmic solutions and software paradigms.

**Author Contributions:** Conceptualization, B.U. and A.S.; Methodology, B.U. and A.S.; Software, B.U.; Formal analysis, B.U. and A.S.; Investigation, A.S.; Resources, B.U. and A.S.; Writing—original draft preparation, B.U. and A.S.; Writing—review and editing, B.U. and A.S.; Visualization, B.U. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The data presented in this study are included within the article in the form of tables.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Dijkman, R.M.; Dumas, M.; Ouyang, C. Semantics and analysis of business process models in BPMN. *Inf. Softw. Technol.* **2008**, *50*, 1281–1294. [\[CrossRef\]](#)
2. Paolone, G.; Marinelli, M.; Paesani, R.; DiFelice, P. Automatic Code Generation of MVC Web Applications. *Computers* **2020**, *9*, 56. [\[CrossRef\]](#)
3. Durai, A.D.; Ganesh, M.; Mathew, R.M.; Anguraj, D.K. A novel approach with an extensive case study and experiment for automatic code generation from the XMI schema Of UML models. *J. Supercomput.* **2022**, *78*, 7677–7699. [\[CrossRef\]](#)
4. Yongchareon, S.; Liu, C.; Zhao, X.; Yu, J.; Ngamakeur, K.; Xu, J. Deriving user interface flow models for artifact-centric business processes. *Comput. Ind.* **2018**, *96*, 66–85. [\[CrossRef\]](#)
5. Idrees, M.; Aslam, F. A Comprehensive Survey and Analysis of Diverse Visual Programming. *Vfast Trans. Softw. Eng.* **2022**, *10*, 47–60. [\[CrossRef\]](#)
6. Mythily, M.; Valarmathi, M.L.; Durai, C.A.D. Model transformation using logical prediction from sequence diagram: An Experimental approach. *Clust. Comput.* **2019**, *22*, 12351–12362. [\[CrossRef\]](#)
7. Zafar, I.; Azam, F.; Anwar, M.W.; Maqbool, B.; Butt, W.H.; Nazir, A. A Novel Framework to Automatically Generate Executable Web Services From BPMN Models. *IEEE Access* **2019**, *7*, 93653–93677. [\[CrossRef\]](#)
8. Núñez, M.; Bonhaure, D.; González, M.; Cernuzzi, L. A model-driven approach for the development of native mobile applications focusing on the data layer. *J. Syst. Softw.* **2020**, *161*, 110489. [\[CrossRef\]](#)
9. Sunitha E.V.; Samuel, P. Automatic Code Generation From UML State Chart Diagrams. *IEEE Access* **2019**, *7*, 8591–8608. [\[CrossRef\]](#)
10. Tragatschnig, S.; Stevanetic, S.; Zdun, U. Supporting the evolution of event-driven service-oriented architectures using change patterns. *Inf. Softw. Technol.* **2018**, *100*, 133–146. [\[CrossRef\]](#)
11. Apostol, D.; Rusovan, P.; Marcu, M. UML to code, and code to UML, a view inside implementation challenges and cost. In Proceedings of the 26th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, Romania, 19–21 October 2022; pp. 140–145. [\[CrossRef\]](#)
12. Sánchez-Morales, L.N.; Alor-Hernández, G.; Rosales-Morales, V.Y.; Cortes-Camarillo, C.A.; Sánchez-Cervantes, J.L. Generating educational mobile applications using UIDPs identified by artificial intelligence techniques. *Comput. Stand. Interfaces* **2020**, *70*, 103407. [\[CrossRef\]](#)
13. Chaber, P.; Ławryńczuk, M. AutoMATiC: Code Generation of Model Predictive Control Algorithms for Microcontrollers. *IEEE Trans. Ind. Inform.* **2020**, *16*, 4547–4556. [\[CrossRef\]](#)
14. Uyanık, B.; Şahin, V.H. A Template-based Code Generator for Web Applications. *Turk. J. Electr. Eng. Comput. Sci.* **2020**, *28*, 1747–1762. [\[CrossRef\]](#)
15. Bocciarelli, P.; D’Ambrogio, A.; Panetti, T.; Giglio, E. MDAV: A Framework for Developing Data-Intensive Web Applications. *Informatics* **2022**, *9*, 12. [\[CrossRef\]](#)
16. Dinkelbach, J.; Razik, L.; Mirz, M.; Benigni, A.; Monti, A. Template-based generation of programming language specific code for smart grid modelling compliant with CIM and CGMES. *J. Eng.* **2023**, *1*, e12208. [\[CrossRef\]](#)
17. Sebastián, G.; Tesoriero, R.; Gallud, J.A. Automatic Code Generation for Language-Learning Applications. *IEEE Lat. Am. Trans.* **2020**, *18*, 1433–1440. [\[CrossRef\]](#)
18. Ding, J.; Lu, J.; Wang, G.; Ma, J.; Kiritsis, D.; Yan, Y. Code Generation Approach Supporting Complex System Modeling based on Graph Pattern Matching. *IFAC-PapersOnLine* **2022**, *55*, 3004–3009. [\[CrossRef\]](#)
19. Hu, K.; Duan, Z.; Wang, J.; Gao, L.; Shang, L. Template-based AADL automatic code generation. *Front. Comput. Sci.* **2019**, *13*, 698–714. [\[CrossRef\]](#)
20. Yang, G.; Zhou, Y.; Chen, X.; Zhang, X.; Han, T.; Chen, T. ExploitGen: Template-augmented exploit code generation based on CodeBERT. *J. Syst. Softw.* **2023**, *197*, 111577. [\[CrossRef\]](#)
21. Anuar, A.W.; Kama, N.; Azmi, A.; Rusli, H.M. Revisiting Web Application Development with Integrated Records Management Important Aspectusing Re-CRUD. *J. Inf. Knowl. Manag.* **2022**, *12*, 31–54.
22. Sunitha, E.V.; Samuel, P. Object constraint language for code generation from activity models. *Inf. Softw. Technol.* **2018**, *103*, 92–111. [\[CrossRef\]](#)
23. Akbulut, A.; Toprak, S. Code generator framework for smart TV platforms. *IET Softw.* **2019**, *13*, 268–279. [\[CrossRef\]](#)
24. Possatto, M.A.; Lucrédio, D. Automatically propagating changes from reference implementations to code generation templates. *Inf. Softw. Technol.* **2015**, *67*, 65–78. [\[CrossRef\]](#)

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.