DOI: 10.15439/2023F5691
ISSN 2300-5963 ACSIS, Vol. 37

Can ChatGPT Replace a Template-based Code Generator?

Adam Bochenek
0009-0005-1259-3418
National Information Processing Institute
al. Niepodległości 188b, 00-608 Warsaw, Poland
Email: adam.bochenek@opi.org.pl

***Abstract*—This article examines whether a large language model (LLM) tool, such as ChatGPT, can replace a template-based source code generator. To this end, we conducted an experiment in which we attempted to replace an existing template-based DAO class generator (which creates entity classes and a repository for a specified database table) with a solution in which templates of target classes were presented to ChatGPT alongside the source model. We then instructed ChatGPT to generate new classes. A novelty in this work is an attempt at two-stage cooperation with ChatGPT: first we provide the pattern, then we fill it. The experiment proved that, at present, such a solution yields results that are neither predictable nor replicable, and successive attempts to execute the same commands returned wildly varying results. ChatGPT randomly recognises the rules that are present in templates, and complex instructions impact the generated results negatively. At present, classic code generation methods yield markedly superior results.**

I. Introduction

THIS article aims to determine whether, and if so, to what degree, large language model (LLM) can work as a source code generator [2]. Can the manual creation and subsequent population of a template be replaced by an LLM tool, such as ChatGPT, which has been taught to read patterns and treat them as templates to be populated with specific data?

Software engineers strive to ensure that the degree of abstraction in which they operate is as close as possible to the concepts that are present during the stages of application analysis and modelling. This approach streamlines work, increases productivity, and reduces the number of potential errors. Typically, this involves increasing the degree of abstraction and applying concepts related to a specific domain directly and as broadly as possible during the software development stage.

One method of achieving this goal involves the application of a model-driven development (MDD) methodology, such as domain-specific modelling (DSM). The essence of this methodology lies in its capability to model and specify the target application at an abstraction degree that caters to the needs of experts and analysts who are familiar with the given domain [1]. When such specification is completed, the final product (i.e. application code) should be generated automatically. The manner in which the code is generated is the focal point of our study.

The findings of this article should be considered an experiment. We examine whether the classic method of code generation based on available templates can be replaced by an LLM-based tool (for the purposes of this article, we used ChatGPT in its 2023, Mar 14 version).

In this work, we first list the areas of knowledge that will be of interest to us, i.e. Model Driven Development, code generation and the use of LLM in the field of programming. Then we move on to the description of the SDSM method, the Osfald tool and the experiment itself, which will consist in using ChatGPT as a template-based code generator. The most important point is the description of the results of the experiment and the conclusions drawn from it.

II. Related work

A. Model-driven development, domain-specific modelling, and template-based code generators

Model Driven Development (MDD) is a set of application development methodologies in which models are used as the basis of the entire software development process. It involves creating a model that describes the complete system, or a fragment thereof, which is then used as a base for generating source code (source model -> source code).

In MDD, models are typically created using formal notation, such as unified modelling language (UML), business process model and notation (BPMN), or domain-specific language (DSL). These models are then used in automated code generation. The advantages of MDD include the capability to use domain-specific concepts during the design stage, automated software production, increased effectiveness, higher quality, and streamlined change management. The weaknesses include the high cost of implementation and mandatory specialist knowledge of formal languages.

DSM is a variant of MDD. In DSM, models are created to describe specific domains. We use languages that are specific for the given domain and serve as the basis for the generation of the code and other project artefacts. DSM's chief strength lies in domain-specific languages and models typically being easier to understand than general-purpose ones [1].

The DSM approach comprises three key elements:

- the model
- the code generator
- the framework.

Template-based code generation (TBCG) is a method of automated source code generation based on templates or patterns [14], which are powered by models. In the TBCG approach, software developers define source code templates that contain particular variables or parameters; subsequently, such templates are populated with specific values to create the source code for the given project.

Despite its obvious advantages, such as exceptionally quick generation of source code and improvement of the code's quality by minimising errors caused by manual typing, TBCG also entails its own set of flaws. The most notable include:

- the complexity of code template creation and management
- the complicated handling of complex design problems that require a more algorithmically advanced approach
- necessary knowledge of template language and associated software development libraries.

With these flaws in mind, we examined whether the TBCG method could be replaced with the capabilities offered by LLMs.

B. Codex, Copilot, GPT-3

Rapid progress in the creation and use of LLMs has created new opportunities for automation of the software development process. The current approach, domain-level modelling and automated transformation into source code [5], [6], [7], [8], has been supplemented with methodologies that utilise machine learning and LLMs [9], [10], [11]. New codes are created in response to commands formed in natural language, or as attempts to supplement or complete existing code.

Both approaches have their flaws. Due to their nature, DSL languages match specific domains, and will never become general-purpose tools; generative LLMs have difficulty extracting complex coding patterns from code corpora, and often generate codes riddled with syntax or semantic errors [12], [13]. The results returned by either model are seldom predictable or replicable.

The experiment described in this article attempted to combine both approaches. We wanted the code generated to correspond to the specified pattern, and to be predictable and replicable. With consideration for the complex and time-consuming nature of template creation, we attempted to substitute it by providing an LLM with an example or a set of examples to be used as a pattern, which could then be modified after the provision of a new, different set of parameters.

Our experiment is different from the typical use of ChatGPT as a developer helper. Usually, ChatGPT is supposed to generate the source code in a given programming language based on the given natural language prompt.

C. Code generation vs. security

While analysing the methods of automated code generation, we must not omit one crucial aspect: security. Language models and tools that are based on them, such as GitHub Copilot, have been trained on tremendous quantities of open source code. This code contains errors, so the concern that the code suggested by Copilot may potentially contain errors that affect application security is a valid one. The experiment described in [4] demonstrated that in a trial that covered eighty-nine scenarios in which Copilot was used to generate 1,689 applications, as many as 40% of them contained security vulnerabilities.

We believe that a template- or pattern-based method is a significantly safer solution. When creating a template, we can verify its safety; the code generated on the basis of a safe template will also be safe, in most cases.

III. METHODS

A. The simplified domain-specific modelling (SDSM) method

At OPI PIB (National Information Processing Institute - National Research Institute), we have developed our own, in-house application development method. It is based on the DSM approach, but is simplified. We call it simplified domain-specific modelling (SDSM). As with DSM, this method is also based on three key elements:

- the model
- the code generator
- the organisation- and domain-specific environment (framework).

The perception of the first component, the model, differs from that of the classic DSM. We treat the model as input data for the code generator. We neither require nor define any formal language that describes the solution at a higher degree of abstraction. We do not define any rules or syntaxes. We do not use DSL at all; instead:

- we create simple models that are understood as embedded at the level of the data structure domain
- we search for existing models. We often discover that raw or processed data, which can act as a model (input data) for the code generator, already exists.

In the SDSM method, application generation comprises four stages:

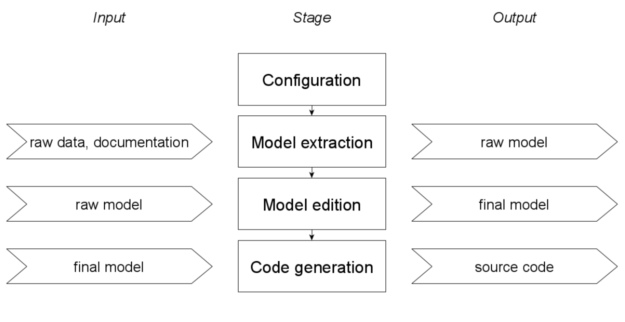

Fig. 1. SDSM stages and input and output artifacts.

Stage one, **configuration**, is an auxiliary step which enables all operations that prepare the development process to be conducted properly. This might involve specifying the place from which the data that serves a model will be read, establishing a link to a database, or specifying where documentation is stored. This step is optional. During stage two, the **model extraction** stage, we read information from an external source that has been configured in the previous step and use it to build the model. Stage three is **model edition**: the model obtained during stage two may require modifications or additions. Stage three may also be used to create a new model from scratch if there is no source from which the model can be obtained. The data structure used to generate the code is created and edited during this stage. Stage four involves **source code generation** based on the model prepared during the previous stages. Stages (and their input/output artifacts) are shown in Fig. 1.

B. The Osfald tool

OPI PIB's SDSM method is implemented by the Osfald tool, an application that acts as a framework, and offers ready-to-use, universal components and functions that are required to create code generators. Osfald is also a set of interfaces that are a recipe for an SDSM-type generator. Here, a generator is understood as an implementation (a set of classes that implement created interfaces) that enables us to progress through all stages of application development; in other words, it extracts and edits the model, and generates new code based on the model. Generators may pertain to different application elements and offer various degrees of complexity. Defining a new generator chiefly involves implementing such previously-developed interfaces.

C. TBCG-type DAO generator vs. LLM

One of the best and most complete examples that demonstrates the SDSM concept in action is the generation of the data access layer for a typical business application written in Java. Although implementation details differ depending on the libraries used, this layer typically handles two basic class types: entity classes and repository classes. On the application side, the entity class represents one row in a database table. Its primary component is a field list. The repository class is a set of methods that includes basic methods that correspond to the create, read, update, and delete (CRUD) functions, complemented with additional functions used to search for entities in accordance with specific criteria.

The generator fulfils the following tasks (by SDSM stage):

1) Configuration: in this case, establishing a connection to the database
2) Model extraction: by using standard JDBC mechanisms in Java, we read the structure details of selected database tables (field names, their types, lengths, and requirements)
3) Model edition: for each table field, a corresponding entity field is generated that bears a default (albeit editable) name and type. During stages two and three, a data model, which acts as input for the generator, is created
4) Code generation: based on previously-defined templates and the model prepared during stages two and three, entity source code and a repository (and, optionally, a test class) are generated in the specific part of the application. The existing TBCG-based generator uses the Apache Velocity engine and templates to produce code.

Example 3.1: **EnGptUser** as an example of an entity class.

```
public class EnGptUser {
  private long idAuto;
  private String idUid;
  private String firstName;
  private String surname;
  private Integer age;
}
```

Example 3.2: **RepoGptUser** as an example of a repository class.

```
public class RepoGptUser extends BaseRepo
 ↪  {
  public RepoGptUser(Trx trx)
  public void create(EnGptUser en)
  public void update(EnGptUser en)
  public List<EnGptUser> findAll()
  public EnGptUser findByKey(String key)
  private void entity2Stmt(EnGptUser en,
   ↪  PreparedStatement stmt, boolean
   ↪  update)
  protected void rs2Entity(ResultSet rs,
   ↪  EnGptUser en)
}
```

The entity class consists of fields that correspond to database table fields for which a specific entity has been created. The **EnGptUser** class is an example of an entity class (Example 3.1).

The repository class **RepoGptUser** (Example 3.2) is more complex, as it requires a base class that it expands (e.g. a repository that is based on a specific library).

D. Template table, entity, and repository

The experiment described in this article involved attempting to replace the existing template-based entity and repository class generator (the TBCG method) for a specific database table with an LLM-based solution. ChatGPT (Mar 14 version) was the LLM tool used in this test. In place of templates, we prepared a template database table with corresponding template entity and repository class. The table contained all field type variants, and the entity class contained all possible mappings of those variants to Java types. The term 'variants of mapping' refers to the principle according to which table fields that allow null values are mapped to Java object types, while table fields that do not allow null values are mapped to Java primitive types. We apply this principle in the TBCG templates, and we expected the LLM model to detect and apply the principle similarly.

Example 3.3: The **template_table_01** template table.

```
CREATE TABLE template_table_01
(
  id_auto bigint NOT NULL,
  id_uid character varying(32) NOT NULL,
  string_field_a character varying(255)
   ↪  NOT NULL,
  string_field_b character varying(10000),
  text_field_a text NOT NULL,
  text_field_b text,
  bool_field_a boolean NOT NULL,
  bool_field_b boolean,
  ...
  ...
  CONSTRAINT template_table_01_pkey
   ↪  PRIMARY KEY (id_uid),
  CONSTRAINT template_table_01_id_auto_key
   ↪  UNIQUE (id_auto)
)
```

Example 3.4: The **EnTemplateTable01** template entity class.

```
public class EnTemplateTable01 {
  private long idAuto;
  private String idUid;
  private String stringFieldA;
  private String stringFieldB;
  private String textFieldA;
  private String textFieldB;
  private boolean boolFieldA;
  private Boolean boolFieldB;
  ...
  ...
}
```

The template database table **template_table_01** (Example 3.3) contained all field type variants that we wanted our generator to handle.

Class **EnTemplateTable01** (Example 3.4) corresponds to the template table. Each field in this class correspond to a table field. Note that the NOT NULL fields in the table correspond to primitive Java types (e.g. int, long, double), while the places where the database permits NULL values correspond to object types (Integer, Long, Double). We expected ChatGPT to detect and recognise this rule.

The repository class is the most complex. Below, we present one of its key methods, which include mapping the result of SQL query to entity: **rs2Entity** (Example 3.5).

In the **rs2Entity** (and **entity2Stmt**) methods, the most important thing is to match the types correctly (for example: **getInt**, **getIntNull**, **setInt**, **setIntNull**).

The methods responsible for calling the SQL queries are used to add or modify new rows, and to run searches based on criteria entered are the basic repository methods: **create** (Example 3.6), **update**, **findAll**, and **findByKey**.

IV. Experiments

We tested whether TBCG mechanism can be replaced by ChatGPT. This task was divided into two stages: generation of entity classes and generation of repository classes.

A. Stage one: generation of entity classes

Stage one involved attempting to use ChatGPT to generate an entity (a Java class) based on the provided table structure (in SQL). The entity class generated in this way was to correspond to a specific template. First, we provided ChatGPT with the **template_table_01** template table structure and its corresponding **EnTemplateTable01** template entity class. We then asked ChatGPT to use the template to create a new entity for a different SQL structure provided.

To this end, we prepared five different database tables; for each of them, ChatGPT generated an entity. The following criteria were applied to verify the generated entity's correctness and consistency with the template:

- correct class syntax in Java, including compilation readiness
- completeness (whether all fields were included and in the correct order)
- field types (allowing for correct separation into simple and object types, which depends on whether the database allows null values)
- result replicability (whether repeated generation of the entity yields identical source code; three attempts).

When designing the experiment, we ensured that the tables tested would be sufficiently diverse. Our findings are presented below (Table I. Stage 1, entity class generation).

The **Table** column contains the names of the database tables for which an entity class was generated. The **Syntax** column contains information on whether the code generated was correct syntax-wise. For all tables, we received code that could be compiled. The values in the **Completeness** column inform us whether the generated class contained all database table fields. The values in the **Field order** column inform us whether the fields in the entity appear in the same order as in the source structure.

The next two columns pertain to entity field types. The values in the **Field types** column demonstrate whether all generated entity class fields had the expected type that resulted from the template. Inconsistent types appeared during the generation of the class for the **time_and_bool** table. The values in the **Field types (null/not null)** column tell us whether the generated code is divided correctly into simple and object Java types, as defined in the template—which depends on whether the database allows empty field values. The rightmost column shows whether repeated entity generation by ChatGPT yielded identical code. In the last two tables, we observed differences pertaining to field types.

B. Stage two: generation of repository classes

Stage one was completed with moderate success. Although most tasks were completed correctly, some errors occurred. In stage two, the bar was raised higher. Based on the database

Example 3.5: The **rs2Entity** method of the template repository.

```
void rs2Entity(ResultSet rs, EnTemplateTable01 en) {
    en.setIdAuto(StmtGet.getLong(rs, "id_auto"));
    en.setIdUid(StmtGet.getString(rs, "id_uid"));
    en.setStringFieldA(StmtGet.getString(rs, "string_field_a"));
    en.setStringFieldB(StmtGet.getString(rs, "string_field_b"));
    en.setTextFieldA(StmtGet.getString(rs, "text_field_a"));
    en.setTextFieldB(StmtGet.getString(rs, "text_field_b"));
    en.setBoolFieldA(StmtGet.getBoolean(rs, "bool_field_a"));
    en.setBoolFieldB(StmtGet.getBooleanNull(rs, "bool_field_b"));
    ...
    ...
}
```

Example 3.6: The **create** method of the template repository.

```
public void create(EnTemplateTable01 en) {
  executeWrite(
    " insert into public.template_table_01 ( " +
    " id_uid, string_field_a, string_field_b, text_field_a, text_field_b, " +
    " bool_field_a, bool_field_b, int_field_a, int_field_b, long_field_a, " +
    " long_field_b, float_field_a, float_field_b, double_field_a, double_field_b, " +
    " numeric_field_10_4_a, numeric_field_10_4_b, numeric_field_8_2_a, " +
    " numeric_field_8_2_b, date_field, time_field, timestamp_tz_field, " +
    " timestamp_field " +
    " ) values ( " +
    " ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, " +
    " ?, ? " +
    " )"
    ,
    (stmt) -> {
      entity2Stmt(en, stmt, false);
    }
  );
}
```

TABLE I
STAGE 1, ENTITY CLASS GENERATION

Table	Syntax	Completeness	Field order	Field types	Field types (null / not null)	Replicability
the simplest	ok	ok	ok	ok	ok	ok
two ints	ok	ok	ok	ok	ok	ok
lot of numbers	ok	ok	ok	ok	ok	ok
time and bool	ok	ok	ok	ERROR	ok	ERROR
complete reversed	ok	ok	ok	ok	ERROR	ERROR

table structure, we attempted to generate a repository class. This class's methods must ensure correct communication with the database table, and the class itself must also correctly convert the data stored in and read from the table to the values stored in the entity object. Unlike in stage one, three elements must be compatible: the repository class, the entity class, and the database table. Stage two was completed in two steps.

1) Stage two, step one – verification of task feasibility: The purposes of step one were to determine whether ChatGPT was capable of completing this task, and to decide what approach should be adopted. Several variants of this solution were tested (each variant was tested three times):

Variant 1. The first attempt involved applying the method used during stage one to generate entities. In the first prompt addressed to ChatGPT, we defined the structure of the template database table; in the second prompt, we defined the template repository class for the table; in the third prompt, we instructed ChatGPT to generate a repository class for another table with defined parameters. In response to the third prompt, we always received a repository class written in the Python language—despite the template being written in Java.

Variant 2. In this variant, we specified that we wanted the class to be generated in Java. Although the class returned was in Java, ChatGPT ignored the requirement to match the predefined template. The generated class corresponded to typical repository templates found in the literature. ChatGPT ignored the class template that we had provided.

Variant 3. We decided not to ask ChatGPT to generate a new class, but to modify the existing class in such a manner that it handled a table of a different structure. Once again, the class returned was in the Python language and its operation failed to meet our requirements (Example 4.1).

Variant 4. This variant comprised a single, complete prompt that contained the template table, entity, and repository class, as well as the target entity and table. We asked ChatGPT to generate the target repository. We received a Java repository class that was correct, but did not correspond to our model in any way; instead, ChatGPT used a template that utilised the popular Spring library and the JdbcTemplate class (Example 4.2).

Variant 5. In this variant, we changed our approach and stopped trying to generate a complete repository class; instead, we reduced our scope to single methods. We divided the class into its individual methods and attempted to generate the repository class, one method at a time. The order of method generation was important. This variant was the only one that returned satisfactory results that merited more thorough examination and description.

Example 4.1: Class **RepoTheSimplest** in Python.

```
import psycopg2

class RepoTheSimplest:
  def __init__(self, conn):
    self.conn = conn

  def create_table(self):
    cur = self.conn.cursor()
    cur.execute("""
      CREATE TABLE IF NOT EXISTS
       ↪  the_simplest (
        id_auto BIGINT NOT NULL GENERATED
         ↪  BY DEFAULT AS IDENTITY (
         ↪  INCREMENT 1 START 1 MINVALUE 1
         ↪  MAXVALUE 9223372036854775807
         ↪  CACHE 1 ),
        id_uid CHARACTER VARYING(32)
         ↪  COLLATE pg_catalog."default"
         ↪  NOT NULL,
        one_single_field INTEGER,
        CONSTRAINT the_simplest_pkey
         ↪  PRIMARY KEY (id_uid),
        CONSTRAINT
         ↪  the_simplest_id_auto_key
         ↪  UNIQUE (id_auto)
        );
    """)
    self.conn.commit()
    cur.close()
```

2) Stage two, step two: generation of successive class methods: Step two of the stage two was completed by applying Variant 5 in the following way: the entire task of generating the repository class was divided into sub-tasks, each of which corresponded to the generation of one specific method. For each method, we defined separate correctness criteria. All of the operations described below were performed three times (iterations 1, 2, and 3) to study the replicability of the results.

Replicability in our experiments is one of the key features. When we have a template and the data to fill it, we want to make sure that the fill operation gives exactly the same result every time. Without this certainty, the use of templates makes little sense.

Below is the list of the criteria for the correct generation of each method:

1) The **entity2Stmt** method
   a) K1.1 Correct method syntax (Java)
   b) K1.2 Correct field list (names, order)
   c) K1.3 Correct types (including null/not null).
2) The **rs2Entity** method

Example 4.2: Class **RepoTheSimplest** using Spring.

```
@Repository
public class RepoTheSimplest {

  private final JdbcTemplate jdbcTemplate;

  @Autowired
  public RepoTheSimplest(JdbcTemplate
   ↪  jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  public void create(EnTheSimplest entity)
   ↪  {
    String sql = "INSERT INTO the_simplest
     ↪  (id_uid, one_single_field) VALUES
     ↪  (?, ?)";
    jdbcTemplate.update(sql,
     ↪  entity.getIdUid(),
     ↪  entity.getOneSingleField());
  }
```

a) K2.1 Correct method syntax (Java)
b) K2.2 Correct field list (names, order)
c) K2.3 Correct types (including null/not null).

3) The **create** method
   a) K3.1 Correct method syntax (Java)
   b) K3.2 Correct query syntax (SQL)
   c) K3.3 Correct field list (names, order)
   d) K3.4 Correct en parameter type
   e) K3.5 Correct use of entity2Stmt.
4) The **update** method
   a) K4.1 Correct method syntax (Java)
   b) K4.2 Correct query syntax (SQL)
   c) K4.3 Correct field list (names, order)
   d) K4.4 Correct en parameter type
   e) K4.5 Correct use of entity2Stmt.
5) The **findAll** method
   a) K5.1 Correct method syntax (Java)
   b) K5.2 Correct query syntax (SQL)
   c) K5.3 Correct type of the entity list returned
   d) K5.4 Correct use of rs2Entity.
6) The **findByKey** method
   a) K6.1 Correct method syntax (Java)
   b) K6.2 Correct query syntax (SQL)
   c) K6.3 Correct type of the entity list returned
   d) K6.4 Correct use of rs2Entity.

C. *Analysis of the results*

The results are in Tables II, III, IV and V. Stage one, the creation of entity classes, resulted, for all test tables, in the generation of syntactically correct and complete Java classes that contained all fields expected in the target structure. Field order was also correct. However, for one of the classes, ChatGPT selected wrong field types; for another, it misapplied the type selection rule (primitive vs. object) relative to whether the database permits null values. The results of the experiment in stage two were less optimistic. First, multiple attempts were necessary to formulate the commands in a manner that would result in ChatGPT generating a new repository based on the template provided. Eventually, we were forced to compromise: instead of having ChatGPT generate a complete repository class, we opted to have it create the individual methods of the class. This resulted in ChatGPT having to generate these methods in the correct order, because some of them depended on methods created previously. When attempting to create a repository for a very simple structure (the **the_simplest** table), we encountered problems with type matching: in one method, ChatGPT ignored the dependency on the auxiliary method (this issue did not occur again during repeated attempts). We encountered a considerably higher number of errors in the case of the repository for the **complete_reversed** table. This table had the most complex structure—although, compared to the template table, it differed only with respect to field names and field order. A number of the generated methods had incorrect syntax due to erroneously generated field types, and could not be compiled. In the cases of these methods, the generated code's replicability was very low. Summary - ChatGPT:

- does not fully learn the patterns given to it
- does not recognize all the rules contained in the patterns
- confuses programming languages
- the size of the standard causes deterioration of the result
- the code generated on the basis of the pattern is not replicable.

TABLE II
RESULTS FOR THE **THE_SIMPLEST** TABLE (ERRORS ONLY)

Criterion	Result (iter. 1)	Result (iter. 2)	Result (iter. 3)
K1.3	ERROR	ERROR	ERROR
K2.3	ERROR	ERROR	ERROR
K6.4	ok	ERROR	ok

TABLE III
REPLICABILITY OF THE CODE GENERATED FOR THE **REPOTHESIMPLEST** CLASS METHODS

Method	Replicability
entity2Stmt	yes
rs2Entity	yes
create	yes
update	yes
findAll	yes
findByKey	NO

V. CONCLUSION

It seems that at present, LLM-based code generation methods are unable to replace TBCG. ML- and LLM-based tools,

TABLE IV
RESULTS FOR THE **COMPLETE_REVERSED** TABLE (ERRORS ONLY)

Criterion	Result (iter. 1)	Result (iter. 2)	Result (iter. 3)
K1.1	ERROR	ERROR	ERROR
K1.3	ERROR	ERROR	ERROR
K2.1	ERROR	ERROR	ERROR
K2.3	ERROR	ERROR	ERROR

TABLE V
REPLICABILITY OF THE CODE GENERATED FOR THE **REPOCOMPLETEREVERSED** CLASS METHODS

Method	Replicability
entity2Stmt	NO
rs2Entity	NO
create	yes
update	yes
findAll	yes
findByKey	yes

such as ChatGPT, provide us with tremendous, previously-unknown capabilities, and are highly likely to change our current perspective on the software development process. At this juncture, however, we were unable to obtain results that would enable us to replace the classic template-based code generation methods. The primary stumbling blocks are ChatGPT's unpredictability and lack of replicability: successive attempts at generating code based on the same commands can yield wildly varying results. Even when the code presented is correct, successive versions differ with regard to details. These differences occur at various levels. In some cases, it is the way in which the code is formatted; in others, it is differences in how variables and methods are named or in the libraries used in the code. Another issue lies in how ChatGPT handles increased complexity, which is demonstrated by our attempts to generate repository methods. When provided with a template and a simple data structure, ChatGPT used the rules defined in the template correctly; with complex structures, however, the result was flawed. Step one of stage two also demonstrated that the formulation of effective commands demands considerable effort. In our case, it took five attempts, and we had to stop trying to generate complete repository classes and be satisfied with only the individual methods.

However, taking into account that tools such as ChatGPT are only at the beginning of their development path, we should watch them closely and hope that soon, in the next versions, they will meet our requirements and will be able to work as a full equivalent of traditional code generators.

And because at OPI PIB we deal with the topic of automatic application generation, we intend to check and test the possibilities of subsequent LLM tools on an ongoing basis.

REFERENCES

[1] Steven Kelly and Juha-Pekka Tolvanen, "Domain-Specific Modeling: Enabling Full Code Generation," John Wiley & Sons, 2008.

[2] Sven Jörges, "Construction and Evolution of Code Generators," Springer-Verlag Berlin Heidelberg, 2013.

[3] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22, New York, NY, USA, 2022, Association for Computing Machinery. DOI https://doi.org/10.1145/3491101.3519665

[4] Hammond A. Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," 2022 IEEE Symposium on Security and Privacy (SP), pages 754–768, 2021. DOI https://doi.org/10.48550/arXiv.2108.09293

[5] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa, "Syntax-guided synthesis," 2013 Formal Methods in Computer-Aided Design, pages 1–8, 2013. DOI 10.1109/FMCAD.2013.6679385

[6] Allen Cypher, "Eager: programming repetitive tasks by example," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1991. DOI https://dl.acm.org/doi/10.1145/108844.108850

[7] Sumit Gulwani, "Automating string processing in spreadsheets using input-output examples," In ACM-SIGACT Symposium on Principles of Programming Languages, 2011. DOI https://doi.org/10.1145/1925844.1926423

[8] Vu Le and Sumit Gulwani, "Flashextract: a framework for data extraction by examples," Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014. https://doi.org/10.1145/2666356.2594333

[9] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow, "Deepcoder: Learning to write programs," ArXiv, abs/1611.01989, 2016. DOI https://doi.org/10.48550/arXiv.1611.01989

[10] Tonglei Guo and Huilin Gao, "Content enhanced bert-based text-to-sql generation," ArXiv, abs/1910.07179, 2019. DOI https://doi.org/10.48550/arXiv.1910.07179

[11] Shirley Anugrah Hayati, Raphaël Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig, "Retrieval-based neural code generation," In Conference on Empirical Methods in Natural Language Processing, 2018. DOI https://doi.org/10.48550/arXiv.1808.10025

[12] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota, "An empirical study on the usage of bert models for code completion," 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pages 108–119, 2021. DOI https://doi.org/10.48550/arXiv.2103.07115

[13] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 336–347, 2021. DOI https://doi.org/10.48550/arXiv.2102.02017

[14] Eugene Syriani, Lechanceux Luhunu, and Houari A. Sahraoui, "Systematic mapping study of template-based code generation," Comput. Lang. Syst. Struct., 52:43–62, 2017. DOI https://doi.org/10.48550/arXiv.1703.06353