Models Trump Code: An Empirical Analysis of GDPR Compliance

Hoàng Nguyễn
hoang.nguyen@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Srđan Krstić
srdan.krstic@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

David Basin
basin@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Ensuring that modern systems comply with privacy regulations, like GDPR's purpose limitation and user consent, is complex, time-consuming, and error-prone. Recently, a model-driven privacy approach has been proposed, enabling developers to declaratively specify a privacy model from which correct enforcement code is automatically generated. This approach promises to reduce software development complexity and minimize developer errors.

We report on the results of an empirical study with 61 participants comparing model-driven and traditional code-centric approaches to implementing privacy requirements in web applications. The participants were graduate computer science students with relevant software engineering and privacy training, making them good developer proxies. They were tasked with implementing the same web application using both approaches. We evaluated the resulting implementations for correctness, development effort, and maintainability. Our findings provide empirical evidence of the superior efficiency and correctness of the model-driven approach. Despite limited prior experience, participants using the model-driven approach wrote eight times less code and produced applications with 64% fewer privacy violations than the participants using the code-centric approach. This is the first study to empirically analyze and demonstrate the benefits of model-driven approaches for developing privacy-critical applications.

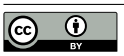
CCS Concepts

• **Software and its engineering** → **Model-driven software engineering**; • **Security and privacy** → *Domain-specific security and privacy architectures*; • **Social and professional topics** → *Software engineering education.*

Keywords

Data protection; privacy engineering; models; runtime enforcement

ACM Reference Format:
Hoàng Nguyễn, Srđan Krstić, and David Basin. 2026. Models Trump Code: An Empirical Analysis of GDPR Compliance. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832656



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832656

1 Introduction

Motivation. Modern information processing systems must satisfy a wide range of privacy requirements stemming from stringent regulations, end-user concerns, self-imposed constraints, and best practices. Implementing these requirements is difficult and error-prone.

The lack of compliance to the General Data Protection Regulation (GDPR) provides evidence of this difficulty. The GDPR (Article 5) requires organizations to protect personal user data from unauthorized access, unlawful processing, and ensure explicit user consent. In practice, organizations frequently fail to meet these requirements. As of December 2025, the most common GDPR fines involve "insufficient legal basis for data processing" (797 fines; €3.01 billion) and "non-compliance with data processing principles" (737 fines; €2.52 billion), with both categories involving violations of user consent and purpose limitation requirements. Overall, nearly 2,700 fines totaling €6 billion have been issued since 2018 [27].

A system must, at a minimum, protect personal data from unauthorized access and modification. Additionally, it must not process the protected data arbitrarily. The system is required to have a *privacy policy*, declared upfront, specifying the purposes for which personal data will be used. Any data usage beyond the declared purposes is deemed unlawful unless supported by another legal basis. Furthermore, consent must be obtained from the owner of personal data prior to its use. Ideally, users should have the option to consent to just specific parts of the privacy policy, allowing them to create a personalized *data-usage policy*, which the system must then enforce.

Focus and problem statement. In this paper, we focus on the most frequently violated data protection requirements, mandating the strict enforcement of users' data usage policies. Recently, Krstić et al. [24] proposed a model-driven privacy methodology that uses system design models to specify these policies and other data protection requirements (Section 2). These design models naturally capture such requirements in a simple and concise manner. They also have a formal semantics that enables the *automatic* generation of a correct-by-design enforcement mechanism. This mechanism instruments the application's actions, tracks actual processing purposes, and prevents actions on personal data from being performed for undeclared purposes or without user consent.

Taking privacy by design literally seems very appealing. Intuitively, applications developed directly from models should be less prone to privacy violations than those developed in a code-centric way. Moreover, their development should be easier, as the tedious enforcement code for data protection requirements is generated automatically. However, there is no prior systematic study supporting this claim and current practices often rely on ad-hoc code-centric solutions. This leads to the following overarching research question:

Do design models improve implementation correctness and reduce development effort for privacy requirements?

Prior work has not provided a clear answer to this question. Krstić et al. [24] implemented a model-driven development tool and used it to conduct three case studies, but did not compare their approach to the code-centric one or conduct a systematic study, as they were the tool's only users. Similarly, other tools proposed for modeling privacy requirements [1, 2, 17, 32] have neither been systematically nor empirically validated.

Our approach. To answer the above question and shed light on why developers struggle to implement privacy requirements, we conducted the first empirical study that tasked participants with implementing privacy requirements. Specifically, our main control variable is the approach choice: we compare a model-driven approach to a traditional, code-centric one. Our study (Section 3) is conducted as part of a project in a graduate security engineering course, where students implemented the same web application using both approaches. Applications using the code-centric approach serve as the control group. The study was conducted in 2024 and was approved by our institution's ethics review board.

Overall, 61 graduate students participated in the study. After a 6-week training, each participant was asked to implement a web application twice: once using a code-centric approach and once using a model-driven approach. The web application was intentionally designed with simple functionality to minimize general requirements engineering effort, while still incorporating common privacy-sensitive features such as advertisements, analytics, statistics, and recommendations (Section 4). To successfully implement the web application in our study, a participant must apply their knowledge across different software development phases: requirements engineering, design, implementation, and evolution. The model-driven approach required more effort in the design phase, whereas the code-centric approach required more effort in the implementation phase. Submitted applications were evaluated using over 800 automatically generated tests, which exercised all privacy requirements and necessary checks. These tests also covered subtle data protection scenarios, such as cases where users provide partial but fine-grained consent. Additionally, students completed an anonymous survey to provide their subjective assessments of the two alternative approaches and their respective tradeoffs.

We analyze the two approaches both quantitatively and qualitatively. Our main findings (Section 5) are:

- Applications developed using the model-driven approach consistently pass more tests (i.e., they are more privacy-compliant) than those using the code-centric one, across all development phases. They have 64% fewer privacy violations on average.
- Manually implementing privacy requirements is challenging, even for proficient participants: no application developed using the code-centric approach passed all tests, whereas several developed using the model-driven approach did.
- The greatest challenge in ensuring privacy is accurately modeling and implementing policies that involve complex purposes and the frequent use of personal data.
- Modeling privacy requirements without generating enforcement code (i.e., relying on *manual* implementation) neither improves development time nor privacy compliance.
- Our survey results indicate a preference for the model-driven approach. This is supported by the development statistics: model-driven development increased time efficiency and significantly reduced development effort, even for students with more experience in code-centric development.

To the best of our knowledge, this is the first empirical study comparing model-driven and code-centric approaches for implementing privacy requirements (Section 6). All the study material is publicly available in our artifact [25]. We hope that our well-designed study, web application, and our findings will serve as a benchmark for future research on privacy requirements (Section 7).

2 Preliminaries

Development typically consists of five phases: requirements engineering, design, implementation, verification & validation, and evolution. We introduce below the two approaches to implementing privacy requirements that we compare in this paper: a model-driven approach with the *v*ActionGUI (*v*AG) tool (Section 2.1), where development takes place in the design phase, and a code-centric approach with the Flask web framework (Section 2.2), where development occurs in the implementation phase. Both approaches are also used in the evolution phase of our study.

2.1 Model-driven Approach

Model-driven privacy [24] specializes model-driven development [3, 7, 23] to privacy. It connects system implementations with privacy requirements using a design model that expresses data protection requirements. This model has precise semantics, defined in terms of the allowed system executions. It uses semantic-preserving model transformations to generate code checking every system action and suppressing those disallowed by the modeled requirements.

The *v*ActionGUI (*v*AG) tool implements a model transformation that takes data, security, and privacy models as input and produces a web application with a complete, configured enforcement mechanism for the access control (AC) requirements specified by the security model and data protection requirements specified by the privacy model. While AC requirements are relevant for data protection (Art. 5 §1(f) GDPR), we omit the security model from our presentation, as its effectiveness has already been demonstrated [4].

Data model. A data model is a 6-tuple $(C, E, attr, A, end, meth)$ that defines the well-formed system states and corresponds to a (simplified) UML class diagram, i.e., to a set of classes C and enum types E with attributes $attr(c)$, for each $c \in C \cup E$, and optionally, methods $meth(c)$, for each $c \in C$. Classes may also be related by binary associations $A \subseteq C^2$. Each association $a = (c, c')$ has a pair of association ends $end(a) = ((el, ml), (er, mr))$ where the left end $end(a)[l] = (el, ml)$ belongs to c, and the right end $end(a)[r] = (er, mr)$ belongs to class c'. Each association end is a tuple consisting of names el and er and multiplicities ml and mr (0..1 or 0..∗).

Example 1. The data model corresponding to the black part of the UML class diagram in Figure 1 (page 7) includes four classes, $C = \{$`Person`, `Event`, `Category`, `Ad`$\}$, one enum $E = \{$`Role`$\}$, and seven associations. All associations have multiplicities 0..∗, except the one between the `Person` and `Event` classes, where each `Event` object can be associated with at most one `Person` object as the event's owner. There are six methods in total. For example, the `Event` class defines the `view_event()` and `edit_event()` methods.

A data model induces a set of actions $\mathcal{A}$:

$$\{\text{create}(c), \text{delete}(c) \mid c \in C\} \cup$$
$$\{\text{read}(c,n), \text{update}(c,n) \mid n \in \text{attr}(c),\ c \in C,\ \text{or}$$
$$\exists c'.\ (n, 0..1) = \text{end}((c,c'))[l] \text{ or } (n, 0..1) = \text{end}((c',c))[r]\} \cup$$
$$\{\text{read}(c,n), \text{add}(c,n), \text{remove}(c,n) \mid$$
$$\exists c'.\ (n, 0..*) = \text{end}((c,c'))[l] \text{ or } (n, 0..*) = \text{end}((c',c))[r]\}.$$

For example, the action create(Category) creates a new Category object, update(Category, name) sets its name, and add(Category, events) adds an event to its association end.

We use the *Object Constraint Language* (OCL) [16] to define properties of a data model. Every OCL expression evaluates to a value of some type. We denote by θ_t the set of OCL expressions that evaluate to instances of type t. Expressions evaluating to Boolean (θ_{Bool}) are called OCL constraints. OCL's dot operator is used to access object attributes (e.g., c.name), or association ends (e.g., c.events). The arrow notation invokes operations on collections (e.g., c.events→**size**()). When applied to a collection, the dot operator maps over its elements. OCL expressions can also be written in the context of an object, which can be referenced using the reserved keyword **self**.

Example 2. The following OCL constraint, defined in the context of the Event class, restricts the possible Event objects to those whose owner is always one of its managers, and whose managers are all attendants: **self**.managedBy → **includes**(**self**.owner) and **self**.managedBy→**forAll**(m |**self**.attendants→**includes**(m)).

Privacy model. Let $(C, E, attr, A, end, meth)$ be a data model. *v*AG requires the designer to identify one class $c \in C$ to represent the application's users. The class must then have an attribute role $\in attr(c)$, whose type is an enum $\mathcal{R}$. For example, in Figure 1, the Person class serves as the user class, with the role enumeration $\mathcal{R} = \{\text{Visitor}, \text{Regularuser}, \text{Moderator}, \text{Admin}\}$.

Personal data is $\mathcal{PD} \subseteq \bigcup_{c \in C}\{c\} \times attr(c) \cup \{(c, end((c,c'))[l]) \mid (c,c') \in A\} \cup \{(c', end((c,c'))[r]) \mid (c,c') \in A\}$ i.e., a subset of attributes and association ends. Let $\mathcal{P}$ be a set of purposes and let $\preceq_{\mathcal{P}}$ be a partial order on $\mathcal{P}$ modeling the purpose hierarchy.

Example 3. Let $\mathcal{PD}$ = {(Person,name), (Person,gender)} be personal data and $\mathcal{P}$ = {Marketing, Functional, Recommendations, Core, Any} the set of purposes. We define $\preceq_{\mathcal{P}}$ as the smallest partial order satisfying: Recommendations $\preceq_{\mathcal{P}}$ Functional, Core $\preceq_{\mathcal{P}}$ Functional, Functional $\preceq_{\mathcal{P}}$ Any, and Marketing $\preceq_{\mathcal{P}}$ Any.

For a fixed data model, a privacy model is defined as a 5-tuple $(\mathcal{PD}, \mathcal{P}, \preceq_{\mathcal{P}}, \mathcal{DP}, \mathcal{AP})$, where $\mathcal{PD}$ is the set of personal data, $\mathcal{P}$ is the set of purposes, $\preceq_{\mathcal{P}}$ represents the purpose hierarchy, $\mathcal{DP} \subseteq \mathcal{PD} \times \mathcal{P} \times \theta_{\text{Bool}}$ is the *privacy policy*, and $\mathcal{AP}$ is a partial function from $meth(c)$, for $c \in C$, to $\mathcal{P}$, mapping methods to purposes. Each element $(d, p, \varphi) \in \mathcal{DP}$ is a declared purpose, stating that personal data d may be used for purpose p when the OCL constraint φ holds. The actual purpose mapping $\mathcal{AP}$ defines which purpose each method is associated with during system execution.

Example 4. Consider the privacy policy "*We use your name for the Core purpose, if you are a regular user.*" This can be formalized as ((Person, name), Core, **self**.role = Regularuser) $\in \mathcal{DP}$. Below is a privacy model, in *v*AG syntax, specifying the personal data and purposes from Example 3, along with the declared purpose corresponding to this policy.

```
1  Personal data {
2    Person.name, Person.gender
3  }
4  Purposes {
5    CORE, RECOMMENDEVENTS, MARKETING,
6    FUNCTIONAL includes CORE RECOMMENDEVENTS, ANY includes MARKETING FUNCTIONAL
7  }
8  Actual purposes {
9    Event.view_event CORE
10 }
11 Declared purposes {
12   Person.name CORE [self.role=REGULARUSER] <you are a regular user>
13 }
```

*v*AG supports a textual description (enclosed in angle brackets) of an OCL constraint. This description is used to automatically generate the privacy policy text shown to users in the web application. Furthermore, the privacy model specifies that the actual purpose of the view_event() method is Core (line 9).

Based on the models, *v*AG automatically extends the data model and uses the extension to generate a web application with a runtime enforcement mechanism for data protection. It generates a stub to be implemented for each method in the data model, e.g.,

```
1 def view_event(id):
2     pass
```

The data model extension (Figure 1, left) captures the state of user consent during execution. The enforcement mechanism consists of a consent collection interface (i.e., a page where users can grant or revoke consent for each declared purpose in $\mathcal{DP}$), a purpose tracking mechanism (i.e., code maintaining a set of purposes given the currently executing methods from $\mathcal{AP}$), and instrumentation code that intercepts every action from $\mathcal{A}$ to check data protection compliance.

Intuitively, the instrumentation ensures that whenever an action on personal data is executed within the context of a set of active actual purposes: (1) each actual purpose is a declared purpose on that data with a constraint that evaluates to true, and (2) the data subject (i.e., the user who owns the personal data) has consented to the declared purposes.

2.2 Code-centric Approach

We exemplify the code-centric approach with the *Flask* [11] web framework. It binds URLs to Python functions, called *endpoints*, that implement the functionality provided upon visiting the URLs. To do so, an endpoint is decorated (using the Python syntax @) with the route(*url*) function of a Flask object specifying the URL string as a parameter.

To manage the application's state and database access, Flask relies on the SQLAlchemy library [30]. It synchronizes the representations of the objects in memory and the corresponding data in the database tables for a subset of classes marked as persistent. Developers can thereby manipulate objects while the library performs the appropriate SQL commands. For example, to fetch an Event object with an *id* of 3, a developer can write Event.query.get(3).

Each endpoint returns an HTML page shown to the user. Flask builds the page from a template t containing static and dynamic parts. The latter are computed and inserted by the render_template (t, ps^*) function from the provided parameters ps.

Example 5. Assume the Flask object app, the persistent class Event, and the function render_template() are in scope. The following Python code implements a simple endpoint.

```
1 @app.route('/view_event/<int:id>')
2 def view_event(id):
3     event = Event.query.get(id)
4     return render_template('view_event.html',{'event': event})
```

Calling the endpoint `/view_event/3` fetches the event with $id = 3$ (line 3) and returns an HTML page describing it (line 4). The `render_template()` function accesses the event's attributes (e.g., `title` and `description`) and the association ends (e.g., `owner`) to create the HTML page specified by the `view_event.html` template.

The Flask-User [44] library provides user authentication. Like in *v*AG, the developer marks one class to represent the application's users. When a Flask endpoint is called by a previously authenticated user, the library initializes an object called `current_user` with the appropriate instance of the marked class.

Example 6. Consider a modified version of the `view_event()` function that enforces the privacy policy from Example 4.

```
1 @app.route('/view_event/<int:id>')
2 def view_event(id):
3     event = Event.query.get(id)
4     c = Consent.query.filter_by(user = event.owner, purpose = CORE, data =
          ↪ event.owner.name).first()
5     if not c or event.owner.role != Role.REGULARUSER:
6         event.owner.name = ''
7     return render_template('view_event.html',{'event': event})
```

Lines 4–6 restrict access to the event owner's personal data, i.e., their `name`. This restriction will be lifted only if the data subject (i.e., the event owner) has given consent to have their `name` used for the Core purpose (first disjunct of the `if` statement) and has the role Regularuser (second disjunct).

The implementation would require consent checks for each event attendant if the template also were to render attendants' names (see Example 7). Besides consent checking, one must reason about the context in which the `view_event()` function would be called to ensure that Core is always the actual purpose. While this may seem straightforward in this example, in more complex web applications, endpoints may invoke each other under different (actual) purposes.

3 Study Design

We define evaluation criteria for implementing privacy requirements (Section 3.1) and state the research questions we aim to address (Section 3.2). Our study tasks students with implementing a web application as part of a course project (Section 3.3).

3.1 Evaluation Criteria

To assess whether a privacy requirement has been implemented correctly, we consider three criteria.

Privacy policy adequacy. This criterion assesses how reasonable the declared purposes in a privacy policy are. In general, its definition is vague and difficult to operationalize: the relationship between personal data and its purpose is considered adequate *when it cannot be criticized* [15]. In practice, evaluating privacy policy adequacy often relies on informal justification and consensus-based manual review. We take advantage of the fixed nature of the application being developed in our study to establish a set of declared purposes in the master solution. This master solution is defined using *v*AG and treated as the ground truth. We compare this ground truth to the set of declared purposes stated in the students' submissions. A privacy policy in a submission is *adequate* when all declared purposes are included in the master solution. If more declared purposes are present, we manually review them to assess their reasonableness. Conversely, any purposes missing from a student's submission but present in the master solution are identified through testing.

Purpose limitation. This criterion requires that any action performed on personal data must be explicitly declared in the privacy policy. This is narrower than the unlawful processing principle (Art. 5 §1(b) GDPR). We assess this criterion automatically via differential testing between the students' submissions and the master solution.

Consent checking. This criterion requires obtaining explicit consent from the data owner before performing any action on personal data, in line with the GDPR's explicit user consent (Art. 7 §1). In our study, this criterion can be tested automatically, as the consent collection mechanism is provided to the students in both approaches.

Note that given a correct privacy model, *v*AG always fulfills the *purpose limitation* and *consent checking* criteria.

3.2 Research Questions

To compare the two approaches to implementing privacy requirements, we answer the following research questions:

- **RQ1** To what extent does modeling privacy improve an application's implementation of purpose limitation?
- **RQ2** To what extent does modeling privacy improve its policy adequacy?
- **RQ3** To what extent does modeling privacy improve consent checking?
- **RQ4** Which approach is more time-efficient in implementing privacy requirements?
- **RQ5** Does prior experience in designing (respectively, implementing) the same privacy requirements improve their implementation (respectively, design)?
- **RQ6** What are the most challenging aspects of privacy to design (respectively, to implement)?
- **RQ7** How much effort is needed to evolve a privacy policy using a model-driven approach compared to a code-centric approach?

The main controlled variable in our study is the development approach: model-driven or code-centric. Our study participants are tasked with using both approaches as outlined below. We evaluate their implementations' *correctness* with respect to the evaluation criteria above by testing and analyzing declared purposes. We evaluate the required *development effort* based on the lines of code written and the time reported by the participants.

3.3 Project Organization

Table 1 presents the course project timeline. Each row corresponds to a project phase, with columns showing the initial input, required deliverable, and phase duration. The project has five consecutive phases: a 6-week Training (Training) phase, two 2-week Development phases (DevI and DevII), a 1-week Evolution (Evol) phase, and a 2-week survey. All work is completed individually.

During the Training phase, students work through tutorials and lab assignments [25] to build a common foundation in Flask, *v*AG, and GDPR privacy requirements. This material is additionally reinforced through lectures and exercise sessions.

Table 1: Project schedule

Phase	Input	Deliverable	Duration
Training	*Training material*	*Solutions* (optional)	6 weeks
DevI	*Requirements*		
(Group 𝔸)	*TemplatevAG*	*v*AG privacy model	2 weeks
(Group 𝔹)	*TemplateFlask*	`project.py`, (`*.py`)	
DevII			
(Group 𝔸)	*TemplateFlask*	`project.py`, (`*.py`)	2 weeks
(Group 𝔹)	*TemplatevAG*	*v*AG privacy model	
Evol	*ChangeRequest*	*v*AG privacy model	1 week
	TemplateUpdates	`project.py`, (`*.py`)	
Survey	*Questionnaire*	Anonymous answers	2 weeks

Table 2: Project participation summary

Phase		Group 𝔸 ($n = 40$)		Group 𝔹 ($n = 41$)		Total ($n = 81$)	
		n	%	n	%	n	%
DevI	*Students*	34	(85.0%)	27	(65.9%)	61	(75.3%)
	• *vAG*	34		-		34	
	• *Flask*	-		27		27	
DevII	*Students*	32	(80.0%)	25	(61.0%)	57	(70.4%)
	• *vAG*	-		25		25	
	• *Flask*	32		-		32	
	Pairs	32		25		57	
Evol	*Students*	34	(85.0%)	25	(61.0%)	59	(72.8%)
	• *vAG*	33		23		56	
	• *Flask*	33		25		58	
	Pairs	32		23		55	
Survey		12	(30.0%)	11	(26.8%)	23	(28.4%)
	Total pairs					112	(69.1%)

In DevI, students were divided arbitrarily into two evenly-sized groups, 𝔸 and 𝔹. Group 𝔸 was tasked with securing the *v*AG application, while Group 𝔹 was tasked with securing the Flask application. In DevII, the two groups switched roles, with Group 𝔸 securing the Flask application and Group 𝔹 securing the *v*AG application. The results from these phases are used to answer **RQ1**–**RQ3**.

The reason for dividing students into groups is twofold. First, we wanted to ensure that all students' questions about the project description (i.e., requirements engineering) are addressed simultaneously for both application versions in the first phase. Otherwise, if all students used a single approach in this phase, some requirements might diverge across the two implementations, making them incomparable. Second, switching the groups in the second phase allows us to answer **RQ5**.

Both groups received the same *Requirements* document with the application's functional and privacy requirements (Section 4.1), and the initial *v*AG (*TemplatevAG*) and Flask (*TemplateFlask*) code (hereafter referred to as templates). Based on the document, the students were tasked with analyzing the privacy requirements and then modifying the respective templates to implement them. Both templates contain a `project.py` file with endpoints implementing only the application's *functional* requirements (as in Example 5), allowing students to focus on the privacy requirements. More specifically:

***v*AG template** Students also received a fixed data model and an empty privacy model, i.e., with no personal data or declared purposes. They were to modify and submit only the privacy model.

Flask template Besides the implementation of the functional requirements, the `project.py` file included a consent collection mechanism consisting of endpoints for adding, revoking, and viewing `Consent` class instances and a corresponding page for users. The mechanism is configured via an additional, initially empty, `PRIVACY_INPUT` dictionary in `project.py` in which declared purposes should be written. These also need to be enforced, so students must add necessary privacy checks in the endpoints (e.g., like in Example 6). If students choose to organize their code into additional `.py` files, those files must also be submitted.

Students using the code-centric approach are allowed to use other Python libraries and tools (e.g., generative AI) that they deem useful. If necessary, they must submit an updated `requirements.txt` file outlining any new dependencies. This freedom makes this group representative of the current state of practice. Moreover, both groups start from existing unfamiliar code (i.e., the initial template) representing the realistic brownfield coding scenario [9].

In DevII, the groups received the remaining templates and implemented the other version of the application. After DevII, we could answer **RQ5**, i.e., we assess the impact of the prior experience in implementing (resp. designing) privacy requirements in DevI on the design (resp. implementation) in DevII.

In the Evol phase, we released the *ChangeRequest* document with new functional and privacy requirements, and updated the *v*AG and Flask templates (*TemplateUpdates* patch). The students were again asked to analyze the privacy requirements and implement them in both of their applications. The *TemplateUpdates* patch implements functional requirements by only modifying files that the students did not need to submit in earlier phases. Hence, the new changes in this phase could focus solely on the privacy requirements.

We evaluated the students' submissions using a set of unit tests. Examining which tests failed most frequently allowed us to identify the most challenging aspects of the privacy requirements to implement correctly (**RQ6**). We also reviewed each submission's privacy policy to assess its adequacy, i.e., the extent to which it aligned with the given privacy requirements (**RQ2**). Finally, we compared the lines of code across different phases to assess the effort needed to implement and evolve these requirements (**RQ7**). All inputs provided to the students, as listed in Table 1, are publicly available [25].

To conclude the study, we conducted an anonymous survey to collect participants' experiences, opinions, and the time spent on each project phase. The primary goal was to gather data for **RQ4**, which examines the time required to implement and evolve privacy requirements. A secondary goal of the survey was to understand how participants perceive the two approaches at the project's end. In particular, we asked the participants which approach they preferred and why. Appendix A lists the 17 survey questions, which covered the students' prior experience with *v*AG and Flask develop-

ment, the time they dedicated to each project phase, and their personal reflections on both approaches. The survey remained open for two weeks. Email invitations were sent to all enrolled students who had participated in at least one project phase.

The participants have graduate-level coding skills and baseline proficiency in Flask, *v*AG, and privacy, acquired in the Training phase. As the GDPR applies extraterritorially to entities serving EU residents, developers worldwide must implement GDPR provisions; thus, other demographics are less relevant to our research questions.

Table 2 summarizes the participants' attrition per group (shown in columns) and phase (shown in rows), categorized by the tool (*v*AG or Flask). Non-applicable cells are marked with a minus sign (−). The course consisted of 81 students: 40 in Group $\mathbb{A}$ and 41 in Group $\mathbb{B}$. Students were free to choose whether to participate in every project phase, resulting in varying numbers of participants across phases. In the DevI phase, 61 students participated overall. In DevII, four fewer students (two in each group) participated. This yielded 57 pairs of Flask and *v*AG implementations across the two development phases.

In the Evol phase, the four students who participated in the DevI phase resumed participation by submitting the evolved versions of their respective applications. Two students active in previous phases in Group $\mathbb{B}$ did not submit, resulting in 59 participating students in the Evol phase overall. From Group $\mathbb{A}$ in the Evol phase, we received 32 pairs of evolved Flask and *v*AG applications. Of those, 31 were from students who participated in both of the previous phases (as one such student decided to evolve only their Flask application). The remaining pair came from one of the two students who participated only in DevI phase, but decided to submit both *v*AG and Flask applications. We therefore received a total of 55 pairs of evolved Flask and *v*AG implementations.

Finally, 23 students participated in the survey, 12 from Group $\mathbb{A}$ and 11 from Group $\mathbb{B}$, which is 37.7% of the 61 students that participated in at least one phase of the project.

4 Web Application Case Study

We now describe the *Event Platform* web application, developed as part of the course project (Section 4.1). It was designed with simple functionality (to minimize requirements engineering effort) and with non-trivial data protection requirements. These include common privacy-sensitive features such as personalized advertisements, user analytics, event statistics, and recommendations. We also explain how project submissions are evaluated (Section 4.2) and how the course project was conducted (Section 4.3). The students received the same project description as provided in our artifact [25].

4.1 Event Platform

The Event Platform manages events, similar to Meetup or Facebook Events. Anyone can browse the events, whereas registered users can create events, attend or request to attend events, and help manage them. Events are organized into categories, each managed by moderators, and users can subscribe to categories of interest. Administrators have extra management capabilities, such as managing user profiles and viewing event statistics. The platform also displays advertisements and uses personal information to send marketing messages and recommend relevant events to users.

4.1.1 Data model. Figure 1 shows the Event Platform's data model, which defines four classes, namely `Person`, `Event`, `Category`, and `Ad`, and seven binary associations. The `Person` class represents the authenticated platform users and stores their `name`, `surname`, `gender`, `email`, and `role`. The `Event` class stores a `title` and `description`. The `Category` class stores only the category `name`, and the `Ad` class stores only the advertisement `content`.

There are four associations between `Person` and `Event`: a person can own, manage, attend, or request to join events. Each event, in turn, has the association ends `owner`, `managedBy`, `attendants`, and `requesters`. An event is owned by exactly one person, but it can be managed, attended, or requested to join by multiple users. The corresponding association ends in `Person` are `events`, `manages`, `attends`, and `requests`. Each event can belong to multiple categories, represented by the `categories` association end. Each category can include multiple events via the `events` association end. A person can moderate and subscribe to multiple categories, i.e., they are the category's `moderators` and `subscribers`. Conversely, `Person` has the `moderates` and `subscriptions` association ends.

We distinguish between class methods that represent the application interface (e.g., Flask endpoints) and internal methods (shown underlined). Of the 40 methods defined in the model, for simplicity, only six are shown in Figure 1. For example, the `Event` class includes the endpoint method `view_event()` (see Example 5), whereas the `Person` class includes the internal method `recommendations()`.

4.1.2 Privacy requirements. The principle of purpose limitation requires applications to process personal data only for specific, explicit, and legitimate purposes. In this project, we focus specifically on requiring that personal data be used only for the purposes to which the data subjects have explicitly consented. Such high-level principles must, in practice, be refined into system-specific privacy requirements. We therefore derive these privacy requirements so that participants can focus on their implementation.

For *v*AG, students are only required to define the privacy model, as the code infrastructure is generated automatically. To ensure a fair workload across both approaches, we extended the Flask data model's implementation to support the definition of personal data, purposes, their hierarchy, as well as the storage of user consent.

The classes shown in Figure 1 (left) extend the Event Platform data model to support the enforcement of purpose limitation and user consent. The `Purpose` class represents application purposes, each with a `name` and an optional collection of `children` purposes. A basic purpose includes no children, whereas a complex purpose must include at least one. The `PersonalData` class models data considered personal. Each personal data object is linked to an `owner` of the `Person` class. The `Consent` class records the explicit consent given by a `user` (`Person` class) to use their `data` (`PersonalData` class) for a specific `purpose` (`Purpose` class).

Purposes. Figure 2 presents the eight purposes (colored white) specified for the project in the development phases, along with their hierarchy. For example, the Recommendations basic purpose corresponds to the application's event recommendation functionality. The more complex Marketing purpose encompasses all marketing functionality and includes Targeted and Mass basic purposes. At the highest level, the Any purpose represents all functionalities offered by the application.

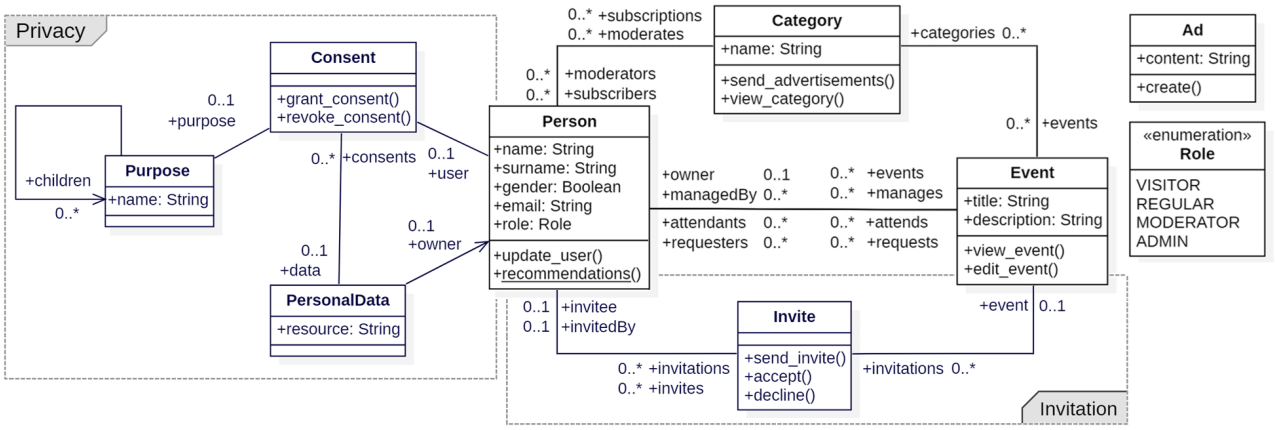


Figure 1: Data model of the Event Platform web application, and its evolution (bottom), with privacy classes (left)

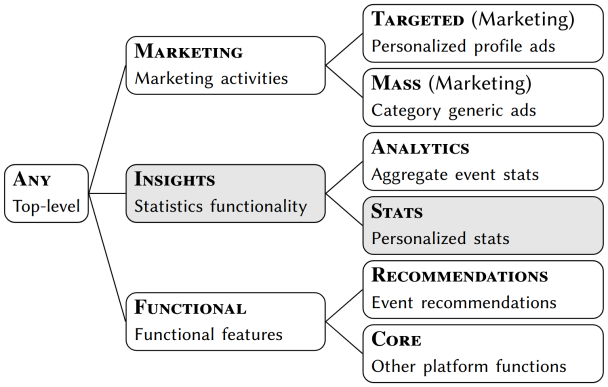


Figure 2: Event Platform purposes and their extension

Personal data. According to Art. 4(1) of the GDPR, any information that can be used to relate or identify a natural person is considered personal data. In our project, all attributes of `Person` and its `subscriptions` association end constitute personal data.

4.1.3 Evolved requirements. In the Evol phase, the Event Platform application additionally allows users to send event invitations to other users. To facilitate this, we extended the application's data model to include an `Invite` class, along with the three associations shown in Figure 1 (bottom). The class is associated with an event (via the `event` association end) and with two users: the user who created the invitation (`invitedBy`) and the user who is invited (the `invitee` association end). The corresponding association ends have also been added to the `Person` and `Event` classes. In addition to providing administrators with aggregate statistics, the application is also extended to offer regular users personalized statistics related to the events they attend, manage, or are invited to, and the categories to which they subscribe. Each user has access to these additional insights via their personal profile page.

Extended purposes. The two new purposes are colored gray in Figure 2. The Insights purpose is a complex purpose including the existing Analytics purpose and a new Stats purpose. The Stats purpose covers the use of user data to provide personalized insights through their profile page, summarizing relevant statistics about their attended events and subscription categories.

Extended personal data. The `Person`'s `attends` association end was considered new personal data. This extension requires students to study the usage of this association end within the application and appropriately adjust the privacy policy.

All these changes were described in the provided *ChangeRequest* document and the appropriate *TemplateUpdates* patch implementing the functional requirements was applied to both Flask and *v*AG implementations of each student.

4.2 Project Evaluation

In this section, we explain how we evaluate students' submissions using two complementary methods. First, we assess their correctness through a comprehensive set of privacy tests that check compliance with *purpose limitation* and *consent checking*. Second, we statically analyze the privacy models to assess their *policy adequacy*. The test summary with examples, as well as our master solution, is provided in our extended report [25].

4.2.1 Privacy tests. We designed a set of tests to evaluate whether students' submissions adhere to the purpose limitation requirement. The test set itself is kept private: it is not revealed to the students prior to their submissions. Our tests cover scenarios where a user has not consented to the use of their personal data for a declared purpose. In such cases, any attempt to use the data for that purpose is forbidden. They also cover scenarios where valid consent has been provided, allowing the data to be used by the application accordingly. We refer to the former scenario as ⟨`None`⟩, and for the latter, we further distinguish between basic (⟨`Basic`⟩) and complex (⟨`Complex`⟩) declared purposes.

As *v*AG tool generates Flask applications, both approaches share the same test set. For the Dev phases, we defined 95 tests each for the ⟨`None`⟩ and ⟨`Basic`⟩ scenarios, and 185 tests for the ⟨`Complex`⟩ scenario. In the Evol phase, we introduced 12 additional tests each for ⟨`None`⟩ and ⟨`Basic`⟩, and 29 ⟨`Complex`⟩ tests. We also re-evaluated each student's updated submission using the test set defined in the Dev phases to detect regression bugs introduced during evolution.

4.2.2 Analyzing privacy models. Our starting point is the master solution: a *v*AG privacy model that adheres to the given project description. The model is both correct (i.e., passing all privacy tests) and adequate (i.e., manually validated by the teaching staff) with respect to the stated privacy requirements.

For *v*AG submissions, we extract the set of personal data, purposes, their hierarchy, and the declared policies directly from the submitted privacy model. For Flask, we extract the corresponding information from the `PRIVACY_INPUT` dictionary. We then compare both artifacts against the master solution. Note that our analysis did not assess whether students had the correct conditions in their declared purposes, as in Flask such conditions are implemented directly in code and cannot be automatically extracted.

Since our master solution declares the minimal set of declared purposes satisfying the privacy requirements, any additional declared purpose in a submission is considered over-declared. While such over-declared purposes do not affect the outcome of the aforementioned privacy tests, they could lead to other violations, e.g., the *data minimization* principle (Art. 5 §1(c) GDPR) stating that personal data should be used only when necessary for a specific purpose.

4.3 Conducting the Project

During the Training phase, students attended three exercise sessions and three lab sessions on *v*AG and Flask. As part of the course's lab sessions, they implemented two *v*AG and three Flask applications. While these exercises were included in the course, participation was not mandatory. To gauge student involvement, we conducted a survey about their participation in these activities.

During the project, collaboration and code sharing were prohibited. However, students were permitted to discuss their testing strategies, for example, by sharing their test sets to improve the quality of their solutions. Moreover, in the code-centric approach, they could use any external libraries or frameworks they considered helpful for correctly enforcing privacy requirements. Students were encouraged to ask questions via a Moodle forum dedicated to the project or to ask the teaching assistants during exercise and lab sessions. Forum discussions were visible to all students.

Student performance was evaluated using the privacy test suite at the project's end. The final grade was based equally on results from both model-driven and code-centric approaches. This ensured that their incentives did not favor either of the two approaches.

5 Analysis

We first present our data (Section 5.1) and then answer our research questions (Section 5.2). We additionally provide some general remarks (Section 5.3) and discuss threats to the validity of our study (Section 5.4). As the data is complex and multidimensional, in our extended report [25] we provide a more precise account of the plots.

5.1 Collected Data

Table 3 displays an overview of students' performance throughout the project. Each row corresponds to a specific phase (i.e., DevI, DevII, or Evol) and shows the total number of valid submissions (i.e., submissions without compilation errors), followed by the average number of failed tests for each test scenario (i.e., ⟨`None`⟩, ⟨`Basic`⟩, or ⟨`Complex`⟩) and tool (i.e., *v*AG or Flask). Given a specific phase, tool, and test scenario, the average is computed by dividing the total number of failed tests by the number of submissions. For the Evol phase, we separately report, in the last row, the average number of failed tests for the newly introduced Evol requirements.

Table 3: Average failed tests by submissions and phases

Tests	Phase	Submissions		⟨None⟩		⟨Basic⟩		⟨Complex⟩	
		*v*AG	Flask	*v*AG	Flask	*v*AG	Flask	*v*AG	Flask
Base	DevI	34	27	32.91	40.15	9.71	20.48	35.18	92
	DevII	25	32	23.32	41.22	1.8	27.72	13.32	110.19
	Evol	56	58	27.2	45.24	5.34	27.52	18.3	96.76
Evol	Evol	56	58	7.5	8.07	3.09	3.83	7.64	11.4

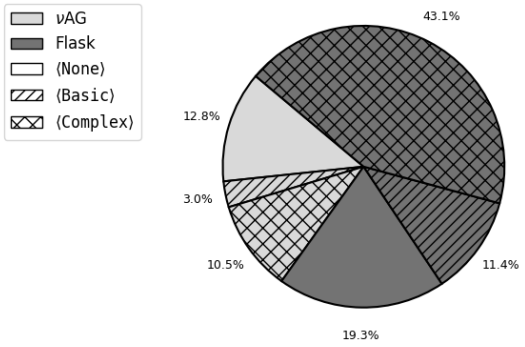


Figure 3: Failed test ratio across all submissions and phases

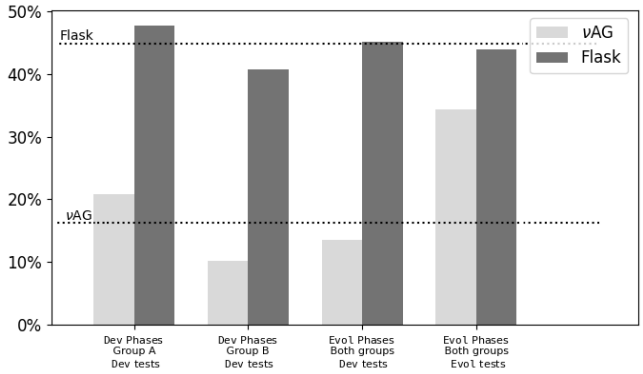


Figure 4: Average percentage of failed tests by phase (····· shows the average percentage of failed tests across all phases)

Figure 3 shows the ratio of failed tests between *v*AG and Flask, aggregated across submissions and phases. Each colored slice is further divided into the three test scenarios.

Figure 4 shows the average percentage of failed tests by phase. In the Dev phases, we separate submissions into groups ($\mathbb{A}$ and $\mathbb{B}$). In the Evol phase, we combine the groups, but separately show the results for the Evol tests. The percentage is calculated by dividing the number of failed tests of each submission by the number of appropriate tests and then averaging this ratio over all submissions in the appropriate group and phase. The two dotted lines represent

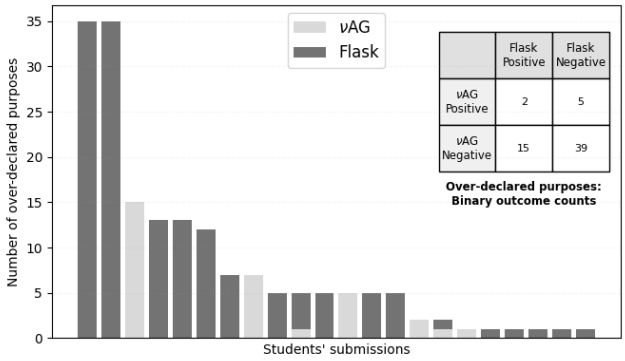


Figure 5: Over-declared purposes in Dev phases

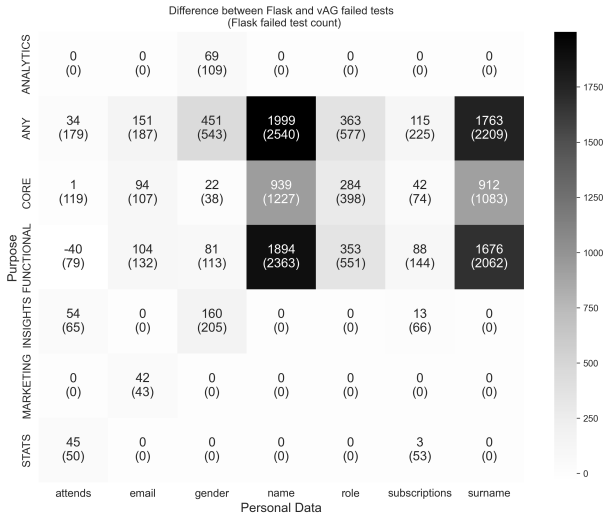


Figure 6: Most violated privacy policies in Dev phases

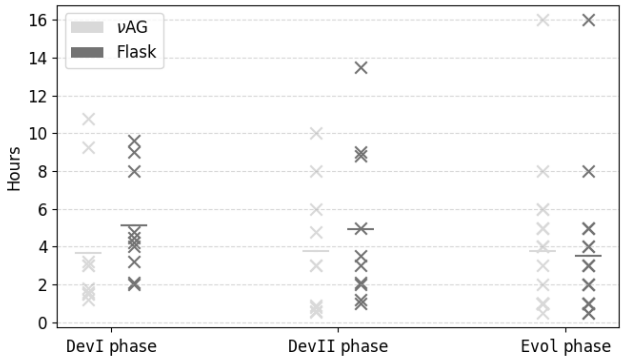


Figure 7: Distribution of the self-reported implementation times ('—' shows the hours on average for implementing each phase)

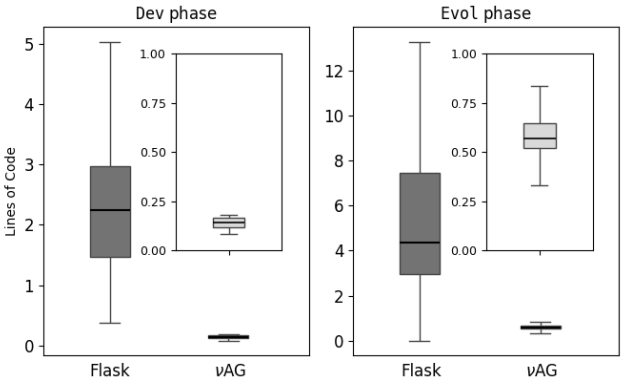


Figure 8: Distribution of the average number of LoC per passed test ('—' shows the median LoC per passed test)

the average percentage of failed tests for *νAG* and Flask across all phases. These averages are calculated by dividing the total number of failed tests by the total number of test runs across all submissions.

Figure 5 shows the number of over-declared purposes found in student submissions during the Dev phases. Each bar represents a student's submission; there are 22 students whose submissions contain this type of violation. The y-axis indicates the number of over-declared purposes in each approach. The table within the plot shows the number of students that over-declared at least one purpose in each tool. Indeed, out of 22 students who over-declared purposes, 15 did so only in Flask, 5 only in *νAG*, and 2 in both tools.

Figure 6 shows a heatmap of combinations of (declared) purposes and personal data most frequently violated during the development phases. Each cell shows the difference between the number of Flask and *νAG* failed tests and the number of Flask failed tests in parentheses, all aggregated over the submissions in the Dev phases.

Figure 7 summarizes the time distribution for implementing privacy requirements across all phases. The data was collected from the survey responses. We received 23 survey responses, which represent 38% of all project participants. Since the project's task also includes other security requirements, such as access control policies, we calculated the time spent on privacy by asking the participants to estimate the percentage of their total time dedicated specifically to implementing privacy policy (Questions 10–15 in Appendix A). Each marker represents the estimate of one participant. The horizontal line — displays the average number of hours spent in each phase.

Figure 8 shows the distribution of the average number of lines of code (LoC) per passed test per submission. In each box plot, the box shows the 25th and 75th percentiles and the middle line indicates the median. The whiskers are bounded by 1.5 of the interquartile range. For each submission, LoC per passed test is calculated by dividing the number of LoC changed by the total number of passed tests. The LoC changes are measured from standard `git diff` output, where each added line or removed line is counted as one changed line. For the Dev phases, we compare the initial template against the student's submission. For the Evol phase, we compare the student's final submission against the one from the previous phases. In the case of *νAG*, we analyze the submission's privacy model; for Flask, we analyze `project.py` and any additional `.py` files submitted.

5.2 Answers to Research Questions

We now provide the key findings from our study by answering the research questions introduced in Section 3. We conducted statistical tests at the significance level $\alpha = 0.05$, which we strengthen appropriately (Bonferroni-adjustment) when we analyze just a subset of the data. We conducted a Shapiro-Wilk normality test on all collected data (failed tests, over-declared purposes, survey answers, and LoC) and established that none are normally distributed. We therefore rely on non-parametric statistical tests, namely the Wilcoxon Signed-Rank (WSR) test for paired data, e.g., when we consider different implementations developed by the same student, and the Mann-Whitney U (MWU) test on non-paired data, e.g., when we compare two independent student groups. We consider paired data whenever possible as such within-subject comparisons control well for individual differences in ability. In each test outcome, we report on the cardinality of the tested data (n), significance (p), effect size (r), computed median difference (Δm), and its corresponding 95% confidence interval (CI). Median values for non-parametric data are estimated using the Hodges–Lehmann estimator.

RQ1: Does modeling privacy improve an application's implementation of purpose limitation?
Answer: Applications built using the model-driven approach (νAG) have significantly fewer privacy violations than those built manually (Flask). On average, νAG submissions have only one-third as many privacy violations as Flask submissions, making them 64% less prone to privacy violations (Figure 3) across all phases (Figure 4).
Statistical justification: We compare implementations using νAG and Flask paired by student, combining submissions from DevI and DevII phases ($n = 57$) with those from the Evol phase containing both implementations ($n = 55$). The WSR test confirms significance ($p = 1e{-}16, r = 0.792$) with the median difference $\Delta m = 100.5$ between failed Flask and νAG tests $[87.0, 133.0]$ CI, favoring νAG over Flask. As secondary analysis, MWU tests comparing independent samples within each phase also favor νAG (with positive Δm) in all three phases: DevI ($\Delta m = 96.0$ tests $[63.0, 131.0], p = 0.0001, r = 0.499$), DevII ($\Delta m = 149.5$ tests $[111.0, 187.0], p = 1e{-}8, r = 0.754$), and Evol ($\Delta m = 134.0$ tests $[110.0, 160.0], p = 1e{-}14, r = 0.693$). All p-values pass the adjusted significance level $\alpha = 0.05/3 = 0.0167$.

RQ2: Does modeling privacy improve its policy adequacy?
Answer: Students are three times more likely to over-declare purposes with Flask than with νAG. Our analysis (Section 4.2.2) shows that 17 out of 61 students over-declared with Flask compared to only 7 with νAG (with 2 making mistakes with both tools) in the development phases. On average, Flask users declared 6.6 unnecessary purposes, while νAG users declared only 1.5 (Figure 5). Most mistakes resulted from unnecessarily associating personal data with complex purposes, which implicitly propagate data usage to all child purposes. For example, declaring a privacy policy "*We use your gender information for Any purpose*" does not violate the purpose limitation principle. However, it is considered over-declared as it permits the use of gender information for an overly broad purpose beyond what is necessary for the application's functionality. This potentially violates the data minimization principle.
Statistical justification: Considering the number of over-declared purposes per student ($n = 57$) paired by tool, the WSR test shows statistical significance ($p = 0.0347, r = 0.812$) but with $\Delta m = 0.00$ over-declarations $[0.00, 0.00]$. As the medians are both zero, because most students do not over-declare, we further analyze the data as binary outcomes: whether a student over-declared at least one purpose or not (see the sub-table in Figure 5). McNemar's test ($p = 0.0414$) yields odds ratio $or = 3$ $[1.13, 7.94]$, confirming that students are three times more likely to over-declare with Flask.

RQ3: Does modeling privacy improve consent checking?
Answer: Applications built using νAG consistently have fewer consent checking violations than those built with Flask across all three test scenarios (Section 4.2.1). For example, νAG submissions result in an average of 28.84 failed tests for the ⟨None⟩ scenario, compared to 40.7 for Flask in the Dev phases (Table 3). The average numbers of failed tests for the ⟨Basic⟩ and ⟨Complex⟩ scenarios in νAG are notably low because consent checking is automatically triggered by the generated enforcement code, whereas in Flask one must manually implement these checks in the correct locations.
Statistical justification: As the consent collection mechanisms are included in the provided templates (Section 3.3), our evaluation focuses on whether implementations correctly check if consent was provided. We perform a WSR test for each of the three test scenarios with adjusted significance level ($\alpha = 0.0167$). Recall that failures of type ⟨None⟩ ($\Delta m = 10.0$ tests $[8.0, 15.0], p = 1e{-}6, r = 0.486$) occur when consent has not been explicitly given, yet the system uses the personal data. Failures of types ⟨Basic⟩ ($\Delta m = 16.0$ tests $[13.0, 21.5], p = 1e{-}14, r = 0.753$) and ⟨Complex⟩ ($\Delta m = 70.0$ tests $[46.0, 92.0], p = 2e{-}14, r = 0.724$) occur when consent has been granted, but the system does not use the data. In all three scenarios, the median difference between Flask and νAG test failures is positive, favoring νAG.

RQ4: Which approach is more time-efficient in implementing privacy requirements?
Answer: While our data suggests that participants using νAG took on average 3.73 hours to implement privacy requirements compared to 5.03 hours for Flask in the development phases (Figure 7), we cannot conclusively confirm that νAG is more time-efficient.
Statistical justification: The WSR test conducted in the development phase ($p = 0.296$) is statistically insignificant due to the low survey response rate ($n = 23$), preventing us from drawing definitive conclusions about time efficiency.

RQ5: Does prior experience in designing (respectively, implementing) the same privacy requirements improve their implementation (respectively, design)?
Answer: Prior experience with the alternate approach does not significantly affect correctness. Figure 4 shows no correlation with prior experience: in DevII, Group $\mathbb{A}$ performs worse with Flask (despite prior νAG experience) than Group $\mathbb{B}$ did with Flask in DevI (with no prior experience), while Group $\mathbb{B}$ improves with νAG (with prior Flask experience) compared to Group $\mathbb{A}$'s νAG (with no prior experience). The tool choice itself and the group's proficiency better predict correctness than prior experience (**RQ1**). The main consequence is that prior design of privacy requirements, unless used to *automatically* generate implementations or is formally linked to them, provides limited benefit in reducing privacy violations.
Statistical justification: Recall that Group $\mathbb{A}$ used νAG during DevI and Flask during DevII, while Group $\mathbb{B}$ used the approaches in opposite order (Section 3.3). We conduct MWU tests comparing groups

with and without prior experience: for Flask, we compare Group $\mathbb{A}$ at DevII (with prior *v*AG experience) and Group $\mathbb{B}$ at DevI (without prior experience); for *v*AG, we compare Group $\mathbb{B}$ at DevII (with prior Flask experience) and Group $\mathbb{A}$ at DevI (without prior experience). Failed test differences for both *v*AG ($p = 0.1120$) and Flask ($p = 0.0841$) are statistically insignificant. Our data may simply indicate that Group $\mathbb{B}$ is overall more proficient, which non-paired tests like MWU do not control for. We could additionally check how much each student improved in DevII (paired data), but, as we have shown in **RQ1**, the tool choice better predicts correctness.

RQ6: What are the most challenging aspects of privacy to design (respectively, to implement)?
Answer: Students struggled most when dealing with privacy policies involving the Core purpose (directly or via a complex purpose) or policies on frequently used personal data (e.g., a user's name).
Justification: As this is an exploratory research question, we do not conduct statistical tests. We exemplify these common mistakes with an implementation taken from one of the submissions.

Example 7. The following code extends the implementation of the `view_event()` endpoint from Example 5 with privacy checks.

```python
def has_consented(user, purpose, property):
    return (purpose, 'Person', property) in {(c.purpose, c.classname,
        ↪ c.propertyname) for c in user.consents}

@app.route('/view_event/<int:id>')
def view_event(id):
    event = Event.query.get(id)
    # Check for the event owner's consent
    if not has_consented(event.owner, CORE, 'name'):
        event.owner.name = ''
    if not has_consented(event.owner, CORE, 'surname'):
        event.owner.surname = ''
    # Check for each attendant's consent
    for attendant in event.attendants:
        if not has_consented(attendant, CORE, 'name'):
            attendant.name = ''
        if not has_consented(attendant, CORE, 'surname'):
            attendant.surname = ''
    return render_template('view_event.html',{'event': event})
```

Here, privacy checks must account for the actual purpose(s) of the data being accessed. The function `view_event()` is considered part of the core functionality of the platform. Therefore, before displaying personal data, the function must check whether consent has been granted for the Core purpose *or any purpose including it* (e.g., the Functional or Any purposes). A common error in student submissions was checking consent only for the basic (Core) purpose, as shown in the `has_consented()` function definition in the example. Additionally, since the page displays not only the event owner's `name` and `surname`, but also those of all attendants, consent must be checked *individually* for each user object in the attendants list.

Indeed, the most common violations involved frequently used personal data, such as `name`, `surname`, or `role`. It is worth noting that the remaining commonly violated policies are associated with the Core purpose. Unlike the more explicit purposes like Marketing or Analytics, the Core purpose refers to the platform's main functionality that is not explicitly enumerated in the privacy requirements. We hypothesize that this ambiguity likely contributed to many implementation errors of the corresponding privacy policy in Flask. Note that *v*AG mitigates most of these violations (Figure 6).

RQ7: How much effort is needed to evolve a privacy policy using model-driven approach compared to code-centric?
Answer: The model-driven approach (*v*AG) requires significantly less code to evolve privacy requirements compared to manual implementation (Flask), but offers no clear time advantage. During the Evol phase, *v*AG applications require fewer than 0.75 LoC per passed test, while Flask applications require more than 4 LoC per passed test (Figure 8). The *v*AG approach is also more uniform and compact, as evidenced by narrower interquartile ranges. However, Flask applications required an average of 3.53 hours to implement compared to 3.78 hours for *v*AG applications, suggesting no time advantage. We hypothesize this is because 65% of the survey participants evolved *v*AG applications first, and the reported time may include overhead from requirements engineering tasks (Figure 7).
Statistical justification: The WSR test for LoC per passed test confirms significance ($\Delta m = 3.82$ LoC/test $[3.07, 4.91]$, $p = 1e{-}9$, $r = 0.861$), with the same trend holding in previous phases. However, survey data on time is statistically insignificant (WSR, $p = 0.903$), preventing definitive conclusions about time efficiency.

Qualitative insights. We have already provided some qualitative insights in the answer to **RQ6**. Here, we provide additional insights obtained from the answers to the last survey question (Q17 in Appendix A). 22 of 23 respondents answered Q17, and of these, 12 provided substantive justifications. As the responses are in free-text format, we applied inductive qualitative coding [45] to analyze them. The analysis was conducted by two coders, both authors of this paper. Each coder initially coded the responses independently and developed their own codebook and coding results. Disagreements were minor (agreement rate 83.3%) and were resolved quickly, requiring only minor revisions to the codebook, provided in Appendix B.
When asked which approach they preferred, 13 favored *v*AG and 7 favored Flask, with one undecided and one suggesting Rocket [34], a web server framework in Rust that has strong type safety. Many students justified their preference for *v*AG, or the model-driven approach in general, as it is more *structured*, *fast*, and *intuitive*, and hence it should lead to more *consistent results*. Nevertheless, the students also reported limitations of *v*AG, including *lack of testing* and concerns regarding its *scalability* and *generalization*. Others preferred Flask for its greater control and easier customization. Although *v*AG currently targets Python, the underlying model-driven privacy approach can support other programming languages, such as the ASP.NET framework in C# [24].

5.3 General Remarks

According to the survey responses, 12 students initially used the *v*AG tool, while 11 began with Flask. In the first phase, both groups similarly rated the clarity of the project description, with a median score of 3 out of 5. Thus, differences in project performance between the two groups are unlikely to be due to the requirements' clarity.

In the Dev phases, none of the Flask submissions passed all tests, while seven *v*AG submissions passed all the tests. More specifically, as shown in Table 3, Flask implementations more often failed to correctly enforce purpose limitation for complex purposes. In contrast, students using *v*AG mostly failed to specify the Core purpose for the application's main functionality.

Considering only students who submitted solutions in all project phases, the introduction of the new requirements in the Evol phase

led to a 2.23% increase in the average number of failures in the base requirements for Flask's tests. In contrast, the *v*AG privacy models did not incur additional violations; instead, they showed a 6.63% decrease in the average number of failures. Further inspection reveals that none of the *v*AG submissions in the Evol phase violated any existing privacy requirements, whereas 27 Flask submissions introduced regressions, i.e., failed tests that had passed in earlier phases. For both tools, no submission passed all evolution tests.

As this was the first year the *v*AG tool was used in the project, no students had prior experience with it. Although three students had worked with an earlier version of *v*AG in the previous iterations of the course, that version did not support modeling privacy. In contrast, according to the survey, 60% of the participants reported prior experience with the Flask framework. Despite this imbalance, students required less time and implemented fewer lines of code while developing more secure applications using *v*AG compared to Flask.

5.4 Threats to Validity

Conclusion validity. We assessed our sample size adequacy with a power analysis for the WSR and MWU tests. To observe statistically significant ($p < 0.05$) medium effects ($r = 0.64$) with 80% power using a WSR test requires 34 pairs of observations. In our study, we have collected 112 pairs of Flask and *v*AG implementations developed by the same participant. To make a similar observation on non-paired data with the MWU test, one needs two groups of 67 observations, whereas for large effects groups of 30 observations are sufficient. Except for the DevI test, in **RQ1**, all our MWU tests detect large effects; hence our two student groups are sufficiently large. Note that in the Evol phase, students are not divided into groups, allowing us to detect medium effects with MWU as well. The survey response rate is too low to provide a statistically significant answer to **RQ4**. We still report the data as it agrees with the other development effort proxies (e.g., LoC per passed test).

Construct validity. In our study, we evaluate the key criteria (Section 3.1) primarily using the generated privacy test suite and static analysis of participants' submissions against the master solution. The LoC metric is an imperfect proxy for development effort, as it does not capture cognitive load or code quality. Our use of LoC *per passed test* provides a normalized measure of coding effort required to deliver privacy requirements (the intended outcome). Finally, we neither measure our participants' ability to code (as they are computer science graduate students with coding experience) nor their ability to interpret privacy regulations (as our project requirements provide system-specific operationalized privacy requirements).

Internal validity. All project work was completed individually. We found no evidence of collaboration or code sharing in discussion forums or student repositories. We ran a plagiarism-detection tool [38] pairwise on all projects and did not detect cases of code sharing. Nevertheless, we cannot fully exclude informal discussions through side channels. We did not observe any clear indications of purely AI-assisted submissions; however, we cannot fully exclude the possibility of such usage. This was not controlled for, as the code-centric approach was intended to reflect an unconstrained development setting. Development time was self-reported retroactively and may be subject to inaccuracies. Moreover, because the project also included tasks to implement security requirements, the allocation of time and effort by the participants may have been influenced.

External validity. While computer science students are commonly accepted as valid proxies for developers [20, 37, 43], our results may not fully generalize to experienced developers. We take further measures to ensure that the use of students in our study is justified. Namely, prior work shows that students and professionals perform similarly when applying new technologies [37] (e.g., *v*AG), which is the primary task in our study. Differences in motivation and prior knowledge may still affect the study outcomes. The former is mitigated by the graded nature of the project, which provides strong incentives for commitment and leads the students to behave closer to developers in the industrial setting [43]. We mitigate the latter threat by providing extensive training material [25] to establish a baseline level of expertise in both approaches. Furthermore, the multi-week, multi-phase structure of our study simulates real-world development better than a (controlled) single-session setting [19, 39]. Our study design also simulates the common scenario that real developers face: the students must initially navigate unfamiliar existing code (i.e., the provided template) and appropriately modify it to implement the privacy requirements. Afterwards, they must further modify their (now more familiar) code to implement the new requirements provided in the evolution phase. Our choice of Flask as the code-centric baseline is representative: with its extensive library support, Python ranks second in web development and Flask is the most-used Python web framework according to the Stack Overflow Developer Survey [42]. This makes it a realistic choice for developing real-world web applications. Participants using the code-centric approach were allowed to use any library or tool they deemed helpful, making this control group representative of the current state of practice. While alternative model-driven tools exist (Section 6), none generate executable enforcement mechanisms from design models. Nevertheless, evaluating additional frameworks and tools, as well as multi-year, group projects (i.e., multiple participants per submission), and different application domains, could further generalize our findings.

6 Related Work

Model-driven development for privacy. Several model-driven approaches support the specification and analysis of privacy policies [8, 32, 46]. However, these methods do not generate executable artifacts that support the runtime enforcement of specified policies. In contrast, Krstić et al. [24] propose a model-driven approach that enforces privacy policies. Their tool, *v*AG, generates Flask applications that support runtime enforcement of purpose limitation and the collection of user consent. The authors evaluated *v*AG on three case study applications they implemented themselves. They used these applications to assess their approach's *generality* across different programming languages and technologies, the *development effort* (in terms of manually written lines of code), and *runtime overhead*. However, they neither compared their approach to the code-centric approach nor carried out a systematic empirical study.

Privacy policy specification and enforcement. Industrial tools like Cedar [40] and Open Policy Agent [12] support policy specification using domain-specific languages. However, these languages offer no native support for privacy-specific notions such as personal

data, purpose, or consent. Antignac et al. [2] introduce an approach for translating high-level privacy architectures into lower-level, privacy-preserving system designs. Pedroza et al. [32] propose a model-based methodology that supports developers in integrating data protection principles into the software development process. Klein et al. [22] and Hublet et al. [21] present enforcement tools that ensure GDPR compliance at runtime using data tainting and information-flow tracking. Ferreira et al. [10] propose a web development framework that assists developers in specifying data protection policies and enforces them in a cross-cutting manner. Their prototype, RuleKeeper, targets applications built on the MERN stack and is evaluated in a study involving ten graduate students to measure development and debugging effort.

Other efforts focus on different aspects of privacy. For example, Antignac et al. [1] model data minimization policies, while Sánchez et al. [33] address data confidentiality. Hooda et al. [17] propose a logic for formalizing privacy policies and use large language models to extract formulae from policy texts. These approaches are complementary to constructive privacy by design methods used to generate executable enforcement mechanisms, and they have not yet been empirically evaluated against current development practices.

Empirical studies on security. Mai et al. [28] propose a use-case driven modeling method for specifying and validating confidentiality, integrity, and availability requirements, and successfully apply this method in an industrial healthcare project. Roussev [35] compares model-driven and code-centric development approaches in educational settings to measure their impact on programming learning outcomes and shows that model-driven approaches improve students' programming skills and help them manage system complexity as the system evolves. "Build It, Break It, Fix It" [14, 31, 36, 48] is a multi-year contest that empirically evaluates secure coding practices, where participants implement, test, and patch vulnerabilities. The contest provides insights into the types and frequency of security vulnerabilities introduced by developers. More recently, Basin et al. [4] demonstrate the benefits of enforcing fine-grained access control policies using a model-driven approach. Our study focuses on the implementation of privacy requirements and is inspired by Basin et al.'s study, i.e., our project is designed to have simple functionality, yet non-trivial privacy requirements.

Empirical studies on privacy. Prior studies on privacy have focused primarily on three areas: analyzing system compliance with regulations, investigating user responses to consent requests, and identifying challenges developers face in understanding and implementing privacy requirements. Several large-scale measurement studies assess the extent to which websites comply with privacy regulations, such as cookie consent compliance [6, 29], and marketing emails [26]. Our work considers compliance with purpose limitation and user consent for *any* personal data, not only cookies. Singh et al. [41] study consent page designs and show that user preferences are driven by clarity and low cognitive effort rather than specific design choices. Utz et al. [47] further examine how such pages affect user behavior and conclude that design significantly influences user decisions, often resulting in uninformed user consent. In contrast, our study tasks participants with implementing privacy requirements using different approaches.

Bednar et al. [5] interview six senior engineers to understand their motivation to comply with privacy regulations. They find that privacy is largely perceived as a compliance obligation rather than an intrinsic development responsibility. The interview conducted by Horstmann et al. [18] identifies a communication gap between developers, privacy experts, and team coordinators in how privacy requirements are understood and enforced, which hinders the effective integration of privacy into development processes. Franke et al. [13] survey 56 open-source developers and report that GDPR compliance increases development and review effort in these open-source projects due to the difficulty of implementing and verifying compliance.

Horstmann et al. [19] conduct a study where 30 developers complete a five-hour assignment to implement three GDPR-related tasks. Similarly, Senarath and Arachchilage [39] conduct a case study with 36 volunteer developers manually implementing privacy requirements in a healthcare application. Both works conclude that developers lack sufficient knowledge and tool support, leading to difficulties in translating privacy requirements into enforcement. Our empirical study involves a larger number of participants over an extended period (13 weeks) that encompasses all phases of software development, from requirements engineering to evolution, and compares the two approaches.

7 Conclusions

Implementing privacy requirements in modern applications is challenging, even when one is familiar with privacy regulations. This state of practice has been around since the GDPR came into force [27]. We have presented the first empirical study comparing a promising model-driven approach to a traditional code-centric approach to achieving privacy compliance. We focused on purpose limitation and user consent, two critical and commonly violated requirements.

Our findings provide strong evidence that the model-driven approach results in applications that are significantly more privacy compliant and require substantially fewer lines of code to develop, even with limited prior experience using model-driven tools.

Acknowledgments

David Basin was supported in part by the Novo Nordisk Foundation through a RECRUIT Sabbatical Grant NNF24OC0089993. Krzysztof Cybulski helped us improve statistical reporting. Over the past decade, security engineering course assistants at ETH Zurich have contributed to the code templates and training material. Finally, we thank the students who participated in our study and provided helpful insights and comments.

References

[1] Thibaud Antignac and Daniel Le Métayer. Privacy Architectures: Reasoning about Data Minimisation and Integrity. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *10th International Workshop on Security and Trust Management (STM)*, volume 8743 of *LNCS*, pages 17–32. Springer, 2014.

[2] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. Privacy Compliance via Model Transformations. In *2018 IEEE European Symposium on Security and Privacy Workshops, (EuroS&P Workshops)*, pages 120–126. IEEE, 2018.

[3] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, 2003.

[4] David Basin, Juan Guarnizo, Srđan Krstić, Hoang Nguyen, and Martín Ochoa. Is Modeling Access Control Worth It? In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2830–2844. ACM, 2023.

[5] Kathrin Bednar, Sarah Spiekermann, and Marc Langheinrich. Engineering Privacy by Design: Are engineers ready to live up to the challenge? *Inf. Soc.*, 35(3):122–142, 2019.

[6] Ahmed Bouhoula, Karel Kubicek, Amit Zac, Carlos Cotrini, and David Basin. Automated Large-Scale Analysis of Cookie Notice Compliance. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium*. USENIX, 2024.

[7] Alan W. Brown. Model driven architecture: Principles and practice. *Softw. Syst. Model.*, 3(4):314–327, 2004.

[8] Pietro Colombo and Elena Ferrari. Towards a Modeling and Analysis Framework for Privacy-Aware Systems. In *2012 International Conference on Privacy, Security, Risk and Trust, PASSAT 2012, and 2012 International Conference on Social Computing, SocialCom 2012, Amsterdam, Netherlands, September 3-5, 2012*, pages 81–90. IEEE Computer Society, 2012.

[9] Michael C. Feathers. *Working Effectively with Legacy Code*. Pearson, USA, 2004.

[10] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. Rulekeeper: GDPR-aware personal data compliance for web frameworks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2817–2834, 2023.

[11] Flask. Flask Web Framework, 2025. https://flask.palletsprojects.com/en/stable/.

[12] Cloud Native Computing Foundation. Open Policy Agent, 2025. https://www.openpolicyagent.org/.

[13] Lucas Franke, Huayu Liang, Sahar Farzanehpour, Aaron Brantly, James C. Davis, and Chris Brown. An Exploratory Mixed-methods Study on General Data Protection Regulation (GDPR) Compliance in Open-Source Software. In Xavier Franch, Maya Daneva, Silverio Martínez-Fernández, and Luigi Quaranta, editors, *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 325–336. ACM, 2024.

[14] Kelsey R. Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L. Mazurek, Michael Hicks, and James Parker. Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1141–1155. ACM, 2022.

[15] Peter Gola and Dirk Heckmann, editors. *Datenschutz-Grundverordnung, Bundesdatenschutzgesetz: DS-GVO / BDSG*. C. H. Beck Verlag, 2022.

[16] Object Management Group. Object Constraint Language (OCL) 2.4 Specification, February 2014. https://www.omg.org/spec/OCL/2.4/.

[17] Ashish Hooda, Rishabh Khandelwal, Prasad Chalasani, Kassem Fawaz, and Somesh Jha. PolicyLR: A Logic Representation For Privacy Policies, 2024.

[18] Stefan Albert Horstmann, Samuel Domiks, Marco Gutfleisch, Mindy Tran, Yasemin Acar, Veelasha Moonsamy, and Alena Naiakshina. "Those things are written by lawyers, and programmers are reading that." Mapping the communication gap between software developers and privacy experts. *Proc. Priv. Enhancing Technol.*, 2024(1):151–170, 2024.

[19] Stefan Albert Horstmann, Sandy Hong, David Klein, Raphael Serafini, Martin Degeling, Martin Johns, Veelasha Moonsamy, and Alena Naiakshina. "Sorry for Bugging You So Much." Exploring Developers' Behavior Towards Privacy-Compliant Implementation. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *IEEE Symposium on Security and Privacy (S&P)*, pages 1215–1233. IEEE, 2025.

[20] Martin Höst, Björn Regnell, and Claes Wohlin. Using Students as Subjects: A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empir. Softw. Eng.*, 5(3):201–214, 2000.

[21] François Hublet, David Basin, and Srđan Krstić. Enforcing the GDPR. In *28th European Symposium on Research in Computer Security (ESORICS)*, pages 400–422. Springer, 2023.

[22] David Klein, Benny Rolle, Thomas Barber, Manuel Karl, and Martin Johns. General data protection runtime: Enforcing transparent GDPR compliance for existing applications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3343–3357. ACM, 2023.

[23] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison-Wesley, USA, 2003.

[24] Srđan Krstić, Hoang Nguyen, and David Basin. Model-driven Privacy. *Proc. Priv. Enhancing Technol.*, 2024(1):314–329, 2024.

[25] Srđan Krstić, Hoang Nguyen, and David Basin. Models Trump Code: An Empirical Analysis of GDPR Compliance, 2026. Artifact and extended report. https://zenodo.org/records/21695500.

[26] Karel Kubicek, Jakob Merane, Carlos Cotrini, Alexander Stremitzer, Stefan Bechtold, and David Basin. Checking Websites' GDPR Consent Compliance for Marketing Emails. *Proc. Priv. Enhancing Technol.*, 2022(2):282–303, 2022.

[27] CMS Law. Enforcement Tracker, 2026. https://www.enforcementtracker.com/.

[28] Phu X. Mai, Arda Goknil, Lwin Khin Shar, Fabrizio Pastore, Lionel C. Briand, and Shaban Shaame. Modeling Security and Privacy Requirements: a Use Case-Driven Approach. *Inf. Softw. Technol.*, 100:165–182, 2018.

[29] Célestin Matte, Nataliia Bielova, and Cristiana Santos. Do Cookie Banners Respect my Choice? Measuring Legal Compliance of Banners from IAB Europe's Transparency and Consent Framework. In *IEEE Symposium on Security and Privacy (S&P)*, pages 791–809. IEEE, 2020.

[30] Pallets Projects. Flask-SQLAlchemy toolkit and ORM. https://flask-sqlalchemy.palletsprojects.com/en/2.x/, 2023.

[31] James Parker, Michael Hicks, Andrew Ruef, Michelle L. Mazurek, Dave Levin, Daniel Votipka, Piotr Mardziel, and Kelsey R. Fulton. Build It, Break It, Fix It: Contesting Secure Development. *ACM Trans. Priv. Secur.*, 23(2):10:1–10:36, 2020.

[32] Gabriel Pedroza, Victor Muntés-Mulero, Yod Samuel Martín, and Guillaume Mockly. A Model-based Approach to Realize Privacy and Data Protection by Design. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&P Workshops)*, pages 332–339. IEEE, 2021.

[33] Óscar Sánchez Ramón, Fernando Molina, Jesús García Molina, and José Ambrosio Toval Álvarez. ModelSec: A Generative Architecture for Model-Driven Security. *J. Univers. Comput. Sci.*, 15(15):2957–2980, 2009.

[34] Rocket. Rocket Web Framework, 2025. https://rocket.rs/.

[35] Borislav Roussev. Empirical Evidence Justifying the Adoption of a Model-Based Approach in the Course Web Applications Development. *J. Inf. Technol. Educ.*, 2:73–90, 2003.

[36] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build It, Break It, Fix It: Contesting Secure Development. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 690–703. ACM, 2016.

[37] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo Juzgado. Are Students Representatives of Professionals in Software Engineering Experiments? In Antonia Bertolino, Gerardo Canfora, and Sebastian Elbaum, editors, *IEEE/ACM International Conference on Software Engineering, (ICSE)*, pages 666–676. IEEE, 2015.

[38] Saul Schleimer, Daniel Shawcross Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *ACM SIGMOD International Conference on Management of Data*, pages 76–85. ACM, 2003.

[39] Awanthika Senarath and Nalin A. G. Arachchilage. Why Developers cannot embed Privacy into Software Systems? An empirical investigation. In Austen Rainer, Stephen G. MacDonell, and Jacky W. Keung, editors, *22nd International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 211–216. ACM, 2018.

[40] Amazon Web Services. Cedar Authorization Language, 2025. https://www.cedarpolicy.com/en.

[41] Ashutosh Kumar Singh, Nisarg Upadhyaya, Arka Seth, Xuehui Hu, Nishanth Sastry, and Mainack Mondal. What Cookie Consent Notices Do Users Prefer: A Study In The Wild. In *European Symposium on Usable Security (EuroUSEC)*, pages 28–39. ACM, 2022.

[42] Stackoverflow Survey. https://survey.stackoverflow.co/2025/technology, 2025.

[43] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using Students as Subjects - an empirical evaluation. In H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch, editors, *2nd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 288–290. ACM, 2008.

[44] Ling Thio. Flask-User Authentication Library, 2023. http://flask-user.readthedocs.io.

[45] David R. Thomas. A general inductive approach for analyzing qualitative evaluation data. *American Journal of Evaluation*, 27(2):237–246, 2006.

[46] Damiano Torre, Mauricio Alférez, Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. Modeling Data Protection and Privacy: application and experience with GDPR. *Softw. Syst. Model.*, 20(6):2071–2087, 2021.

[47] Christine Utz, Martin Degeling, Sascha Fahl, Florian Schaub, and Thorsten Holz. (Un)informed Consent: studying GDPR Consent Notices in the field. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 973–990. ACM, 2019.

[48] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. Understanding Security Mistakes Developers Make: Qualitative Analysis from Build It, Break It, Fix It. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 109–126. USENIX, 2020.

Generative AI Usage

We have used Claude Sonnet for minor editorial improvements (e.g., for grammar and spelling) and for aid in writing statistical testing code. All generated code was manually checked.

Open Science

Our artifact [25] contains an extended version of this paper and our study resources: *v*AG and Flask training material, project description, and implementation templates. With the provided resources, the study can be independently replicated. Due to data protection restrictions, we cannot share the collected participant data.

Ethical Considerations

Our study was approved by our institution's ethics review board (IRB). We adhere to the ethical principles of the Menlo Report, as well as the conference's data protection guidelines. We have conducted a stakeholder-based ethics analysis and implemented multiple countermeasures to ensure that students (the main stakeholders) were not adversely affected by the study.

Stakeholders. Primary stakeholders include the computer science students participating in an elective graduate course, our research team, and the host institution. Secondary stakeholders are future students, the privacy engineering community, software developers implementing privacy controls, end users of privacy-aware applications, and the academic community.

Potential harms and benefits. We identified three potential harms in this research. First, students may perceive coercion, i.e., feel that their participation could affect their grades and thus violate their autonomy, even though no actual dependency exists. Second, the code analysis could potentially violate privacy by revealing programming skills, work habits, and effort patterns, which may conflict with data protection regulations. Third, students might experience psychological distress if they believe their code quality will be scrutinized publicly beyond the normal grading process.

This research benefits multiple stakeholders. Students learn two privacy implementation approaches, receive comprehensive feedback through over 800 test cases, and gain career-relevant privacy engineering skills. The privacy engineering community receives the first empirical comparison of model-driven and code-centric approaches for privacy. Software developers may benefit from productivity gains and fewer privacy violations. End users may benefit from improved privacy protection through better tools. The academic community gains both a research contribution and access to our open science artifacts.

Mitigations. Coercion is addressed by ensuring that all grading was completed and submitted before the research study began, with survey participation occurring only *after* final grades were issued. Students opt in before taking the *anonymous* survey. This temporal separation eliminates actual dependency and substantially reduces perceived coercion. The voluntary nature of the course itself, which is an elective course, further reinforces student autonomy. Anonymization measures address privacy concerns by prohibiting identity collection when accessing code for analysis and there is no personal data collection. We only collect code unlinked from student identities, and anonymous survey responses. Collected data cannot be linked to individual identities. We report only aggregate statistics, and publish no individual code except for anonymized snippets. Transparent communication addresses potential psychological distress through clear messaging that the study *evaluates tools rather than students*. We provide contact information for the study coordinator, institutional data protection officer, and ethics commission for any concerns.

A Survey Questions

Our survey questions are organized into four categories. The questions include the expected answer format in parentheses.

Background knowledge.

Q1. Have you completed the Object Constraint Language assignment before the project? *(Yes/No)*
Q2. Have you completed the Model-driven security & privacy tutorial with *v*AG before the project? *(Yes/No)*
Q3. Have you completed the Model-driven security & privacy assignment with *v*AG before the project? *(Yes/No)*
Q4. Have you completed the Flask authorization lab assignment before the project? *(Yes/No)*
Q5. Have you had any additional experience with *v*AG or its previous versions before the course? *(Yes/No)*
Q6. Have you had any additional experience with Python and Flask before the course? *(Yes/No)*
Q7. In the first phase of the project, which approach were you assigned to use? *(vAG/Flask)*

Requirements engineering.

Q8. The project requirements were sufficiently clear. *(Strongly disagree / Disagree / Neither agree nor disagree / Agree / Strongly agree)*
Q9. In the evolution phase of the project, which of the two approaches did you choose to secure first? *(vAG/Flask)*

Implementation effort.

Q10. How much time in hours approximately did you take to complete the first phase of the project? *(in hours)*
Q11. What percentage of the total time in the first phase of the project was roughly allocated to implementing privacy requirements (versus security requirements)? *(in %)*
Q12. How much time in hours approximately did you take to complete the second phase of the project? *(in hours)*
Q13. What percentage of the total time in the second phase of the project was roughly allocated to implementing privacy requirements (versus security requirements)? *(in %)*
Q14. How much time in hours approximately did you take to complete the evolution phase of the project in *v*AG? *(in hours)*
Q15. How much time in hours approximately did you take to complete the evolution phase of the project in Flask? *(in hours)*

Approach preference.

Q16. For Flask, did you use an existing library or implement the privacy requirements manually? *(Library/Manual)*
Q17. If you were to code a similar project in the future using one technology of your choice, which approach would you choose? Justify your choice. *(vAG/Flask and a short free-text)*

B Codebook

C1. Positive aspects of *v*ActionGUI.
- a. Structured approach
- b. Consistent results
- c. Automation
- d. Promising, intuitive, clear, and fast

C2. Limitations of *v*ActionGUI.
- a. Lack of documentation
- b. Insufficient training
- c. Scalability
- d. Generalization
- e. No testing support

C3. Project comments.
- a. Unclear requirements