Models for Digital Humanities Tools: Coping with Technological Changes and Obsolescence

Simone Zenzaro

CNR-Istituto di Linguistica Computazionale "Antonio Zampolli"(CNR-ILC), Italy
simone.zenzaro@ilc.cnr.it

***Abstract*—This article highlights the importance of defining models and using modular software for digital humanities tools. We emphasize the need for these tools to adapt to the rapidly changing technology landscape, which has led to many tools having short lifespans. We propose an approach to address software obsolescence by using modular models. This involves designing tools to be easily integrated, scalable, and adaptable to changing technology and user requirements. To illustrate the usefulness of models, we present their application to a real-world use case: the CoPhi Editor web platform.**

***Index Terms*—digital humanities, models, modularity, tools, obsolescence**

I. The curse of obsolescence

The history of the Digital Humanities field, have seen vast proliferation of tools meant to support the research in the Humanities [1]–[8]. It is possible to categorize them into at least two broad groups: *standalone* tools and *web-based* tools. A *standalone* tool has a definite purpose and can usually be used without the need of an internet connection. An example of standalone tool is CollateX [9], [10]. The use of web-based tools needs a web user interface of some sort, typically requiring a web server to deliver this interface to users. An example of web-based tool is the Voyant Tools environment [11]. Currently, most Digital Humanities tools seem to favor web technologies, in the form of websites or applications, thus indicating a trend toward web-based tools. An example of this trend can be found in the field of Digital Scholarly Editions, which are usually accessible as artifacts on the web through a user interface (primarily simple HTML, with some exceptions). These interfaces are generated from typically one or more files encoding the edition in the TEI/XML format [12].

The result of applying digital approaches to humanities tasks is a mixed blessing. On one hand, it is now possible to tackle tasks that were previously unfeasible (for example, because they would take too long to complete manually). An example of this is the application of Handwritten Text Recognition (HTR) [13] for digitizing ancient manuscript texts, allowing for review and correction when the HTR fails to recognize the characters—a less burdensome task compared to transcribing the entire text manually.

On the other hand, technology is plagued by the curse of obsolescence. Indeed, the rapid pace at which frameworks and technology are changing and evolving makes it challenging to keep up with updating the tools. This is especially evident in the realm of web technologies, where an added challenge arises with the maintenance of web servers and the associated financial cost of keeping them operational.

So, although such tools are meant to give access to valuable and interesting research results, or to enable other scholars to conduct their studies leveraging these platforms, a great number of tools have been short lived.

While books have stood the test of time, providing a medium that is not only easy to read and preserve but has also remained relatively unchanged over the centuries, their digital counterparts have faced more challenges. Since digital landscape is characterized by rapid evolution and technological obsolescence, it is important to carefully consider and plan for the long-term viability of these digital tools [14] in order to establish a robust and sustainable strategy for the lifespan of digital tools becomes crucial.

As a consequence, relevant research, previously available on the Web, has been lost or is hardly accessible making web sites as Wayback Machine [15] extremely useful to preserve relatively old snapshots of such resources.

In summary, two reasons for this situation can be identified in:

1) rapid obsolescence of technology;
2) web servers maintenance and availability.

The second scenario may stem from various factors such as long-term funding issues, political decisions, or close connections to the life cycle of a research project. any of these issues are external to the domain in which the tool is being developed, so focusing on them would be ineffective. Thus we opted to focus on addressing the challenges associated with rapidly evolving technology. In our view, the development and adoption of models play a crucial role in mitigating these issues.

As stated in [16] in the Digital Humanities (DH) field, "tool building is not a mere research-independent act to enable data processing. Rather [. . .] an intrinsic aspect of research". We will focus on the importance of models for tools and the role of modularity in the software that implements them. We present a definition for the notion of model and we point out where a model is useful. We discuss as well the role of modularity. Then, we briefly present *CoPhi Editor*, a web-based application for authoring digital scholarly editions, analysing how its design follows a model-based and modular approach. We conclude discussing some of the drawback of adopting a model-based approach.

The GreekSchools project, "The Greek Philosophical School according to Europe's earliest 'history of philosophy': Towards a new pioneering critical edition of Philodemus' Arrangement of the Philosophers" has received funding from the European Research Council (ERC) under the European Union's Horizon 2020, Excellent Science (Grant agreement No. 885222).

II. Models

When it comes to the term *model*, there is a large number of possible definitions depending on the context. Finding a definition that is valid for each context is arduous [17], yet modeling is a core activity [18]–[23] in the Digital Humanities. A model usually is a representation of something that is useful to explore and understand better the nature of a target object or system. Although there are several definitions of the term *system*, e.g. [24], [25], here we focus on *software systems*. In order to use a model for Digital Humanities tools with the goal to tame the technology obsolescence we seek for a definition that adheres to the following criteria:

- clarifies the structure and functionalities of the tool
- provides the needed documentation
- is machine actionable
- is generic enough to be applied at different aspects of the tool

The rationale behind this choice is an effort to deter the development of *disposable* tools and instead promote a more sustainable approach that fosters the creation of dependable and enduring solutions. By prioritizing the development and adoption of models, we aim to establish a framework that not only addresses the immediate technological challenges but also lays the foundation for the long-term viability of digital tools in the field of humanities research. Ultimately, our goal is to foster a culture of resilience within the digital humanities community, ensuring that the tools we create continue to serve scholars and researchers effectively for years to come.

Examining the impact of applying formal methods on the overall quality of software design and development, we borrow the definition of *ground model* from [26] as "the conceptual construct, blueprints of the to-be-implemented piece of real world". In [26] the ground model is related to the precise mathematical definition of the Abstract State Machine (ASM) formal method [27], [28].

In the context of Digital humanities tools development, the ground model is a valuable concept for establishing a development framework. While applying the Abstract State Machine method would be beneficial for accurately understanding and modeling specific parts of the software system that implements a tool, we do not intend to apply it uniformly across all aspects of development. Our emphasis lies in utilizing the ASM method and the ground model as a solid starting point for defining criteria upon which we can evaluate the reliability and resilience to change and obsolescence of a tool and choose the "right" model formalism.

A. The ASM ground model

An ASM ground model is an high-level model for complex real-life systems. With high-level meaning that the description of the model should be understandable by non technical experts. The features of a ground model are:

- *precise*: the model should be accurate enough to describe the system while avoiding the introduction of unnecessary details;
- *simple and concise*: the model should be understandable by both domain experts and system designers and to be manageable for inspection and analysis;
- *abstract (minimal) yet complete*: each semantically relevant feature and all the interactions with the environment are present;
- *validatable*: the model should be falsifiable;
- *with precise semantical foundation* as the basis for reliable tool development and prototyping.

In our case, we loosen the last feature in order to allow for different formalism even though the precise mathematical foundation is not fully achievable.

B. Understanding the domain

The process of building a model is in itself an attempt to understand the domain. Within the Digital Humanities field, this step is of paramount importance for ensuring the accurate realization of a tool. Building a model involves delving deeply into the intricacies of the subject matter, dissecting its nuances, and identifying the key components that will drive the development process forward. By meticulously crafting a model, developers gain invaluable insights into the domain's underlying structures, relationships, and functionalities. This deep understanding lays a solid foundation for designing and implementing digital tools that effectively address the needs and challenges of the humanities field. Moreover, the process of building a model fosters collaboration and interdisciplinary exchange, as scholars and experts from diverse backgrounds come together to contribute their expertise and insights.

Typically, when starting the development of a tool, two primary actors come into play: a humanist and a computer scientist. Building an effective and accurate tool requires that both parties overcome a language and communication problem and a share a common language. This effort is not merely about bridging the gap between disciplines; it involves fostering a collaborative environment where humanists and computer scientists can effectively communicate and align their perspectives.

The model is one way of sharing a common understanding of the domain with the constraint to agree on the meaning of a common vocabulary. In the Domain Drive Design (DDD) [29] approach to software design, such common language is referred to as the *ubiquitous language* and it influences the software even in the way its source code is written.

For humanists, the process involves articulating their tacit knowledge [30], [31]—the implicit understanding and expertise that they possess through years of study and experience, making it challenging to express in explicit terms. On the other hand, computer scientists must capture such knowledge and translate it into machine understandable requirements. It appears how important it is an interdisciplinary collaboration and communication in the pursuit of effective digital humanities tools.

This process could greatly benefit from the involvement of a third figure who acts as a mediator, possessing the ability to comprehend and effectively communicate in both

the respective ”languages” of the humanist and the computer scientist. This intermediary role is embodied by the digital humanist, who serves as a bridge between the two disciplines, facilitating collaboration and understanding.

C. Multilevel models

Thinking about a model as a fixed description of a given tool might be misleading. In our opinion, it is more useful to think the model as a *set of models*, each of them holding the same criteria we are discussing. Talking about a set of models, instead of a single one makes it possible to describe the tool looking at it from different points of view. Different aspects of the tool may require a different modelling approaches to accurately capture their peculiarities.

For instance, we may desire a model to encapsulate the tool's business logic, delineating its operational rules, processes, and workflows.

Similarly, a model for the tool's data architecture serves to depict the structure, relationships, and constraints of the data entities utilized by the tool.

Furthermore, a model for the user interface (UI) offers a visual representation of the tool's interface elements, layout, and interaction patterns. This model assists in designing an intuitive and user-friendly interface that meets the needs and expectations of the users.

Lastly, a model for the tool's infrastructure outlines the underlying technological components, configurations, and deployment architecture required to support the tool's operation. This model ensures scalability, reliability, and performance by addressing considerations such as server architecture, networking protocols, and security mechanisms.

But, how to put together these models?

Certainly, defining a set of models introduces the challenge of orchestrating communication between them.

If we maintain each model on separate, orthogonal axes, we only need to describe the interface between the models — in other words, how each model interacts with the others. So the orchestration is resolved by clear interfaces definition. For instance, consider the UI model and the data model. We must define where and how the UI displays the data.

D. Validate the correctness of the tool

Once the – or more precisely one of several versions of the – model has been prepared in a given formalism, it encapsulates all the behavior, data descriptions, and interactions between components as agreed upon and understood by both the domain experts and the software developers.

Therefore, the model becomes instrumental in verifying that every concept has been precisely and unambiguously described, and that every problem has been effectively addressed. By providing a clear and structured representation of the domain, the model serves as a source of validation for the tool's features. It enables both the domain experts and the software developers to systematically evaluate and verify the tool's functionality against the requirements and specifications outlined in the model.

Without a model, validating whether the developed tool aligns with the initial requirements becomes a challenging task.

Moreover, systematically addressing every aspect of the tool is helpful in handling exceptions, whose impact is often underestimated by humanists, as they can resolve them point by point. However, for a software system to ensure correct behavior, it must encompass each exception comprehensively.

E. Derive an implementation

One might consider the source code as a model of the tool itself, and indeed, it is one. However, the model remains a distinct artifact from the source code used to implement the tool. Moreover the source code is not meant to be easily understandable by domain experts and it is bound to a specific programming language. In this sense, the model can serve as a *blueprint* of any implementation.

As an example, in [32], the technological stack used for developing the tool has undergone two changes already. This serves as an illustration of a common challenge encountered when technology evolves: the tool is often required to change along with its underlying technology.

The evolution of technology introduces a dynamic element to the development process, as new frameworks, libraries, and methodologies emerge, offering improved capabilities or security fixes. While these advancements can enhance the functionality and performance of digital tools, they also necessitate corresponding updates and adaptations.

The need to adapt to changing technology highlights the importance of maintaining flexibility in the design and implementation of digital humanities tools. By adopting a model-based approach, developers can facilitate smoother transitions between different technological environments.

Just as the blueprint of a building is not the building itself but rather the framework upon which the building can be constructed or reconstructed, the model serves as the blueprint of the tool. It provides a structured representation of the tool's conceptual framework, delineating its key components, functionalities, and interactions.

As the source of truth of the tool, the model should be used to derive an implementation. Depending on the formalism of the model, such implementation may be a manual translation to source code or even automatic code generation from model as is the case for UML models [33], [34].

Furthermore, without a model, there is no guarantee that different versions of a tool share the same set of functionalities.

F. Document the tool

In a hardly axiomatizable field such as the Humanities, a model is also a way to document a tool.

Because the model formalizes every concept of the tool, it provides a deeper understanding of the tool's gears than the tool itself. Even if the technology changes, one can adapt or rebuild the tool by looking at its model. This statement holds true even when the people involved in implementing the tool change.

As documentation, the model can also be a good way to explain the tool to anyone. In particular when the model abide to the aforementioned criteria, the model stems from a shared language between domain experts and computer scientists and describes unambiguously all the needed functionalities.

Treating the model as documentation also proves beneficial for tracking the evolution of the tool's requirements through extensions and changes. By documenting the initial requirements and specifications within the model, it is possible to establish a clear baseline for the tool's functionality and scope.

As the development progresses, any extensions or modifications to the requirements can be documented within the model, providing a comprehensive record of the tool's evolution over time. This ensures transparency and accountability in the decision-making process.

Moreover, by maintaining a detailed history of requirement changes within the model, developers can effectively manage the complexity of the development.

Furthermore, the documentation provided by the model can serve as a valuable reference for future iterations of the tool or for similar projects within the domain.

In this way, treating the model as documentation enhances the traceability, transparency, and maintainability of the tool, contributing to its sustainability.

III. Modularity

The concept of modularity is common in computer science and it goes back to the Parnas' notion of *information hiding* [35]. Keeping models modular means to strive for the identification and isolation of sets of features independent of each other. This way it is possible to decompose the domain of the tool into smaller components. Each component deals with a, possibly, easier problem of the entire tool easing its later implementation.

The modular decomposition can be applied at different levels. An example of modular decomposition applied to the architecture of the tool is the choice of a Service Oriented Architecture (SOA) that has been already suggested as suitable for the DH domain [16] and that often translates to the comparison between monolithic and microservices architecture [36]–[38].

Modules must be compositional in order to build upon each other. If a model is modular it means that we can later *refine* the model or change one model with another when the requirements change. Moreover, the software implementation of a module might change more easily without requiring to rebuild the entire system. We will see an example of this in section IV.

With a modular model, it is also possible to change the tool incrementally upgrading the underlying technology. Hence with the model, we also decompose the hurdle of updating an entire software system.

IV. CoPhiE ditor

CoPhi Editor [39] is web-based collaborative and cooperative authoring platform for Digital Scholarly Editions (DSE) that is being developed as part of the ERC AdG 885222-GreekSchools project. It is being developed to support the editorial process for the papyrology DSE of the Philodemus of Gadara's Arrangement of the Philosophers. One of the challenges in this project regards the reconstruction of the text that is extracted from carbonized papyri and usually presents many gaps.

Although CoPhi Editor focuses initially on digital papyrology, it is designed to also support the editorial process of any type of text and is compliant with the de-facto standard format for DSEs making the edited text available in XML/TEI.

The peculiarity of CoPhi Editor is that it tries to bridge the gap between the classic editorial – on paper – workflow and the digital philology practices. This aim is achieved implementing the DSL-based DSE approach. The philologists, following this approach, focuses on the text using the modalities and editorial conventions that they are used to and that are the result of decades or even centuries of established common practises in a particular domain.

The software platform interprets the text as Domain Specific Languages (DSL) [40], [41].

As an example, the Figure 1 shows an excerpt from the diplomatic transcription of a papyrus that follows the editorial conventions the papyrologists are accustomed to when they write the text of their printed editions. The Figure 2 instead is an example of how the text would appear inside the CoPhi Editor platform. The two texts are in ancient greek and present negligible differences. The *diple obelismente* symbol (between last two lines) can't appear as an interlinear symbol in the DSL but the overall encoding via the DSL is pretty close to the printed editions texts.

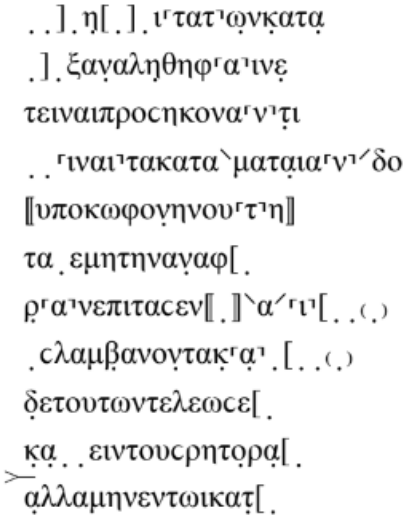

Fig. 1. Editorial conventions.

The resulting DSLs are defined by a precise domain analyses conducted under synergistic collaboration between the domain expert (e.g. the philologist) and the more technical people (e.g. a computer scientist or a digital humanist). Once defined, the DSL captures all of the text phenomena interesting

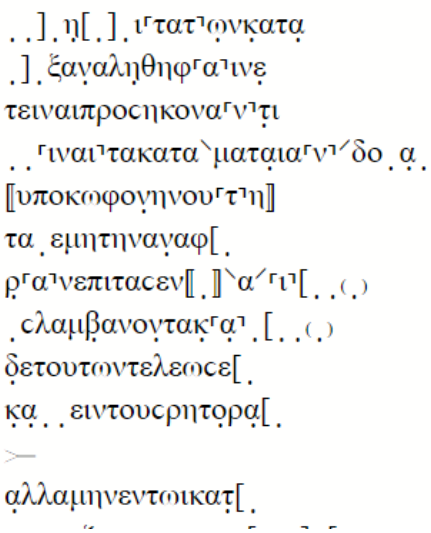

Fig. 2. DSL encoding editorial conventions.

for the domain of application (e.g. the papyrology) inside a context free grammar. An excerpt of the grammar for the diplomatic transcription is shown in Listing 1.

```
// column made of lines
column: line+;
// line ended by a new line char
line: WS* lineNum? WS*
     (text|paragraphos|dipleObelismene)+ NL;
// number of line
lineNum: num WS;
// sequence of digits
num: NUM;
// Latin or Greek text
text: ((latUnit|grcUnit) WS*|num)+;
// extratextual information expressed in Latin
latUnit: latSeq punct?;
// sequence of Latin characters
latSeq: LAT_SEQ;
```

Listing 1. Excerpt of th Contex Free Grammar for the diplomatic transcription DSL

Leveraging the DSL, it is possible to attach automatic or semi-automatic functions to the text, notably check of correctness against the editorial conventions, cross-validation of properties and relations between texts, search capabilities, serialization toward other data formats, text processing, automatic suggestions of conjecture to integrate the missing text (lacunae) leveraging deep neural network such as the one described in [42].

The ability to generate a representation of the DSE in other data formats enables the editors to different scenarios based on how they will publish the DSE. For example, exporting in DOCX the edition is useful to produce a printed edition while exporting to TEI or EpiDoc would be useful for a digital publication.

A. Modular modules in CoPhi Editor

The software architecture of CoPhi Editor was derived by applying modular models to each level of the tool. In this section, as an example, we discuss how the models were applied to the overall architecture of the tool and to the definition of the data for the DSE texts.

Fig. 3 shows the Microservice Architecture of CoPhi Editor.

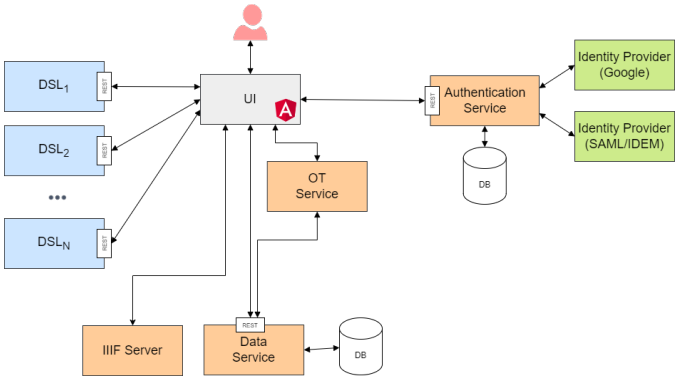

Fig. 3. Example of a figure caption.

This kind of architectural pattern aims to implement a software system as a set of independently deployable, loosely coupled, components. In this particular case, the main modules of CoPhi Editor are:

- UI: the web application component that is the visual representation of the tool, written in Typescript and Angular;
- DSLs: a set of web services that implement each DSL capabilities (e.g. autocompletion and syntax checking) exposing a REST API [43] to communicate with the UI;
- Data Service: a web service exposing a REST API that is in charge of managing the access and manipulation of the DSE data;
- OT Service: a web service that manages the concurrent changes to the text by different scholars making sure that the text integrity is preserved, it also sends push notification to the UI;
- Authentication Service: a web service that enables the authentication to the application with institutional accounts or third part identity providers (e.g. Google, Facebook, etc.);
- IIIF Server: an image server that provides all the needed facsimile images.

The following sections will exemplify how we adopted models for each dimension of the CoPhi Editor platform.

1) UI: The user interface in CoPhi Editor is an Angular[1] web application written in the Typescript programming language[2]. For this kind of applications it is usual to derive the Graphical User Interface (GUI) starting from a set of *wireframes* of *mockups*. At this level a wireframe are an effective model to present an application that does not yet exist. Figure 4 is an example of a *page* of the future web platform where, with simplified lines and sketches, the set of features are laid out before considering visual design elements like color-schemes.

This type of visualisation is easy to grasp for non technical people and is useful to reason about the functionalities of the

[1]https://angular.dev/
[2]https://www.typescriptlang.org/

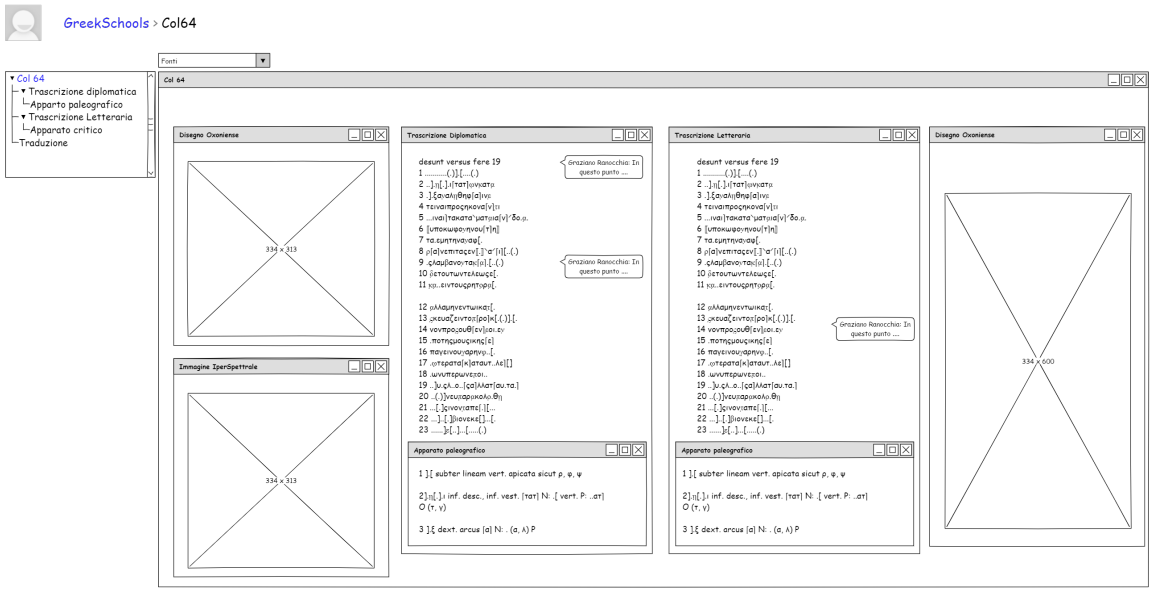

Fig. 4. CoPhi Editor User Interface Wireframe.

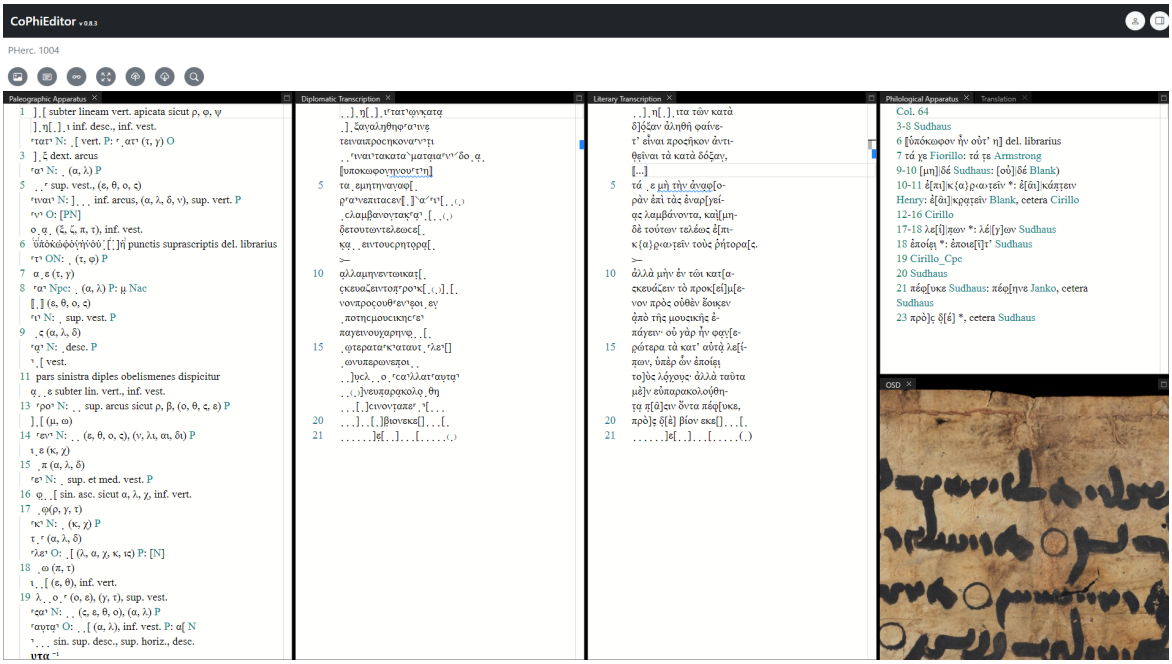

Fig. 5. CoPhi Editor User Interface.

user interface. Changing a wireframe is way easier than modify the code implementing it as a web user interface.

Wireframes also give a good direction for the developer that is able to analyse and identify reusable graphical components before beginning coding them. For instance in Figure 4 the concepts of *window, text editor, facsimile* are recurrent, so the developer can envision the user interface implementing a single component for each concept and putting them together. This type of functionality isolation is indeed an instance of modular design.

Once a first draft for the wireframe is available, the user interface can be developed. Figure 5 shows how the wireframe can become an actual web interface. The differences between a wireframe and the actual web interface can be found in the translation of the simple lines to full fledged web components implementing user interaction, and their visual representation and rendering on screen.

It is conceivable to leverage an additional model as a layer between wireframes and tha actual web interface in the form of *mockups*. A *mockup* would provide a static visual model of

what the user interface will look like in its final form.

Despite the usefulness of mockups, we have chosen to avoid this additional model in order to balance the workload with the resources available to carry out the web platform. This choice gives us an hint about the model based approach to the development of tools: following blindly a model based approach can be counterproductive. It is paramount to always balance effort and benefits when using models.

Leveraging mockups that also describe the functionalities of the web interface enables the tool to be resilient to future technological changes. For example, if the Angular web framework were to no longer be supported (as has already happened with AngularJS), the wireframes would allow for the rebuilding of the entire user interface—or part of it—while keeping the rest of the application (web services, etc.) intact.

Moreover, renovating the technological stack for the user interface can be carried out in parallel with the current web interface maintenance and may even be developed by different individuals than those who originally worked on it.

2) RESTful APIs: Most of the web services in the architecture of CoPhi Editor expose a RESTful Application Programming Interface (API) to access and manage their resources. All of these APIs are defined by a model that follows the OpenAPI [44] specification [3]. This kind of model take the form of a file in JSON[4] or YAML[5] format that is a formal standard to describe HTTP APIs.

Listing 2 shows an example of RESTful endpoint that specifies how to *GET* a text from the data service using the YAML version of OpenAPI.

The advantage of adopting this model manifold. On one hand the specification for a RESTful service in the OpenAPI description provides a single source of truth on how the HTTP calls to the API should be processed.

Such descriptions helps to identify all the possible outcomes in calling the API. For example the specification includes the possibility of server errors or unavailable resources making the discussion about how to manage such “error” cases. Moreover it succinctly describes all the parameters needed to perform a request to the service.

```
/units/{id}:
   parameters:
     - name: v
       in: query
       example: 20231224
       schema:
         type: string
     - name: id
       in: path
       required: true
       schema:
         type: string
   get:
     tags:
       - units
     responses:
       "200":
         description: get the unit
         content:
           application/json:
             schema:
               $ref: "#/components/schemas/unit"
       "404":
         description: Not Found
       "500":
         description: Internal Server Error
         content:
           application/json:
             schema:
               $ref: "#/components/schemas/error"
```

Listing 2. A RESTful endpoint modelled with OpenAPI specification

On the other hand, an OpenAPI specification can be used to generate both the client and the server stubs for the RESTful API. The stubs generated code takes care of checking the formal correctness of the parameters for each service request including authentication, authorization, and security parameters lifting the developer from the boilerplate code to manage the checks.

This means that developers can focus on the actual business logic of each API request and be sure that client and server communicate through compatible interfaces.

3) DSL: The DSL itself can be regarded as a model when referring to its definition through a context-free grammar. The DSL comprises a *lexer* (refer to Listing 3) and a *parser* (refer to Listing 4). The lexer is utilized to convert a sequence of characters into a list of tokens, which are then processed by the parser. Conversely, the parser contains rules used to transform a stream of tokens into an Abstract Syntax Tree (AST), representing the hierarchical structure of the source code’s syntax.

Both the lexer and the parser can be viewed as modular models for the DSLs. An illustration of modularity within such a model pertains to the definition of recognized symbols. For instance, the lexer in Listing 3 defines *L_BRA* as the left square bracket. If the requirements necessitate a change in the representation of brackets to curly brackets, we would solely adjust the lexer while leaving the parser unchanged. Similarly, we can maintain the same lexer and modify the AST structure by updating the parser’s rules.

```
GRC_CHAR: [\u03B1-\u03C9];
L_BRA: '[';
R_BRA: ']';
L_PAR: '(';
R_PAR: ')';
MINUS: '-';
PLUS: '+';
NUM: [1-9];
```

Listing 3. Excerpt of the paleographic apparatus lexer

```
lectio :
   ( GRC_CHAR
   | L_BRA
   | R_BRA
   | L_TOP_HLF_BRA
   | R_TOP_HLF_BRA
   | L_DBL_BRA
   | R_DBL_BRA
   | L_TOP_SML_BRA
   | R_TOP_SML_BRA
   | COMMA
   | SEMICOLON
   | SINGLE_SUB_DOT
   )+
   ;
```

Listing 4. Excerpt of the paleographic apparatus parser

[3]https://www.openapis.org/
[4]https://www.json.org/
[5]https://yaml.org/

4) DSL Service: CoPhi editor is based on Domain Specific Languages. The most common place in which a DSL is used is to encode different kind of texts. The leading idea is to make CoPhi Editor users interact with a programming language without realizing it. That is not meant to trick the users, but to bring the knowledge of Integrated Development Environments (IDEs) – the text editors – in the field of programming languages into the editorial process of Digital Scholarly Editions (DSEs). A Formal grammar defines the syntax of any programming language, and the DSEs are written in a quasi-formal paradigm. Thus it is often possible to generate a context free grammar from the editorial conventions that describe a particular type of text. In the context of CoPhi Editor, the first iteration toward a DSL-based DSE platform has manifested in a collection of markup Domain Specific Languages for papyrology. We have previously discussed how closely such languages can adhere to the actual editorial conventions of the domain (see Figure 1 and 2), however, the scope of what can be managed via a DSL can vary greatly depending on the specific DSE being edited.

The TEI guidelines are de facto standard for DSEs and they find a concrete realization in a specific XML schema that can be actually considered a DSL itself. Therefore, being a DSL-based platform, CoPhi Editor should provide a module that abstracts the interaction with every type of DSL.

All the DSLs services share a common Application Programming Interface (API) – inspired by the Language Server Protocol [6] – that allows to manage all the different languages uniformly inside the CoPhi Editor application. This way it is possible to expand the set of languages available in the platform without requiring any changes to the graphical user interface.

Such API defines a data model and the RESTful endpoint to access syntactical errors checking, contextual suggestions, and syntax highlighting features.

The other model adopted for the DSLs service is the RESTful API definition model.

5) Authentication: The authentication service has been developed adopting an application user model. This model aims to abstract from the authentication provider and describe what a CoPhi Editor user is. In particular, the user model serves the purpose of providing a common interface toward SAML-based authentication [45] and OAuth2-based authentication [46].

We utilize SAML for facilitating authentication via institutional accounts, such as university accounts, which are commonly employed by organizations for secure access to various services and resources. This protocol ensures a standardized approach to authentication, particularly suitable for verifying the identities of users within trusted domains, like educational institutions.

On the other hand, OAuth2 is employed to enable authentication through third-party accounts, including popular platforms such as Google, Facebook, and Twitter. This approach leverages the OAuth2 protocol's capabilities to enable secure and convenient access to our platform using credentials from these external services. By integrating with OAuth2, users can seamlessly authenticate with their existing accounts on these platforms, eliminating the need to create new credentials and enhancing user convenience.

Together, the combination of SAML and OAuth2 provides a comprehensive solution for authentication, catering to both institutional and third-party users. This approach not only ensures the security and reliability of the authentication process but also offers users a flexible and familiar way to access our platform, enhancing user experience and accessibility.

The model defines a user as an authenticated individual for whom two access tokens are generated. The first is a JSON Web Token[7] (JWT) that contains minimal information about the user (i.e., name, surname, and email). This token is meant to be shared among the web services of the CoPhi Editor platform since it is easy to validate the token's integrity and it has a short lifespan –limiting malicious or unintended use of the access token. Since the JWT token has a short lifespan, to grant a work session behaviour for the users, the second token is opaque "refresh" token and is used to automatically generate new JWTs. Since generating new JWTs means that this token can grant access to all of the services on the platform, it must be securely shared between the Authentication service and the web user interface application.

This model is valuable for handling the functionality of each authentication provider separately, ensuring that the authentication process remains modular and adaptable. As long as the generated user adheres to the defined user model of the platform, this approach allows for seamless integration of various authentication mechanisms without compromising the consistency and security of user access. Moreover, by abstracting from the specifics of each authentication provider, the platform can easily accommodate changes or additions to authentication methods in the future.

6) Authorization: While authentication deals with the goal of identify *who* the user is, authorization manages *what* a user can do in the application. In a multi user application, it is usual to define a way to differentiate the users based on the activities that it can perform, namely the *role* of the user. Thus a *role* identifies the set of authorization rules that protects the resources managed by the platform from unintended action.

For instance, the GreekSchools project have three types of users: the editors, the collaborators, and the viewers. The platform defines each of them so that the editor are the only users enabled to change the texts, the collaborators are can only add comments to the texts, the viewers can read texts but have no access to the collaborators' comments and can not change the texts.

The GreekSchools roles are indeed justified for that specific context but, since CoPhi Editor is a domain agnostic platform, such roles should be parametric with regard to the set of DSLs that are currently used for a specific project.

[6]https://microsoft.github.io/language-server-protocol/

[7]https://jwt.io/

The access control in CoPhi Editor is also derived from the definition of a model. In this case the model is a DSL (see Listing 5) used to specify the set of *activities* –as operation on the platform's RESTful APIs – and the *roles* as a set of activities. It is also possible to assign role to specific users. Given these three parts, CoPhi Editor can be configured to the needs of the project context.

```
PostUnits activity can POST /units
GetUnits activity can GET /units
GetNoCommentsUnit activity can GET /units
   where dsl.language is not comment

Contributor role can GetUnits
Admin role can GetUnits, PostUnits
Viewer role can GetNoCommentsUnit

user1 is Admin
user2 is Viewer
user3 is Contributor
```

Listing 5. Example of the authorization DSL

7) Data service: The Data service, as the RESTful web service in charge of storing and managing data shared with the user interface, is the place in which the model for the data arises.

The most important piece of data to be defined and stored are the texts from which the DSE is composed.

The data model is derived from that of [47], which simplifies the definition of text of every granularity and structure by leveraging a recursive text data structure. In the case of CoPhi Editor we refer to *unit* to mean working unit of texts that are associated to a DSL.

V. Downsides of adopting models

There are numerous advantages to developing or embracing models, but there are also associated costs.

Creating a model demands expertise in the subject matter. Moreover, modeling is seldom a straightforward, linear endeavor; rather, it typically entails multiple iterations aimed at revising and enhancing earlier versions until a state of stability is attained.

Beyond the requirement for expertise, developing a model also entails time and resource allocation. Extensive research, thorough requirements analysis, and meticulous design efforts are essential components of the modeling process. Additionally, collaboration among the various actors is necessary to ensure that the model is comprehensive, accurate, and aligned with the specific domain's needs.

Another cost associated with model development is the necessity for validation and verification. This involves conducting rigorous testing, soliciting expert reviews, and comparing the model against real-world data or scenarios to ensure that it accurately represents the system or phenomenon of interest.

Once a software system model is prepared, the process of implementation in a specific technology or programming language can begin. Ensuring alignment between the implemented software and the model is critical, yet achieving this synchronization necessitates meticulous scrutiny of the code and a thorough comprehension of the model.

Selecting the appropriate formalism for each model is a challenging task that is inherently subjective and dependent on various factors. It not only relies on the expertise of the model builder but also on the familiarity of domain experts with that particular formalism.

The choice of formalism involves considering a range of factors, including the nature of the problem domain, the complexity of the system being modeled, the intended use of the model, and the preferences and backgrounds of the people involved. Different formalisms offer distinct advantages and may be better suited to capture certain aspects of the system more effectively.

Another crucial consideration is that while modularity represents a valuable attribute of models, it must be accompanied by the establishment of clear interfaces for effective communication between modules. However, designing effective interfaces requires careful consideration of various factors, including the granularity of module interactions, the level of abstraction, and the specific requirements of the system. Once such interfaces have been defined, the modules have to comply with them.

VI. Conclusions

We have observed that Digital Humanities (DH) is a field in which modeling can be beneficial in the design of new tools. Despite the scientific community acknowledging that modeling is a central aspect of DH research, many tools have had short lifespans.

We have identified two potential reasons for this issue. Consequently, we have chosen to tackle the problem of technological obsolescence by discussing how models can effectively mitigate it in various ways.

We began by introducing the key attributes of a robust model, starting with the concept of a "ground model" within the Abstract State Machine formal method. Moreover, we have highlighted a connection between effective models and modularity.

To illustrate the practical application of models to Digital Humanities tools, we have presented CoPhi Editor, a web-based authoring platform for Digital Scholarly Editions. We have outlined some dimensions in which models have been applied to the tool.

In conclusion, we also discussed some drawbacks of adopting models.

References

[1] I. Pedretti, A. Del Grosso, E. Giovannetti, L. Mancini, S. Piccini, M. Abrate, A. L. Duca, and A. Marchetti, "The clavius on the web project: Digitization, annotation and visualization of early modern manuscripts," in *Proceedings of the Third AIUCD Annual Conference on Humanities and Their Methods in the Digital Ecosystem*, ser. AIUCD '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2802612.2802636

[2] T. Roeder, "Juxta web service, lera, and variance viewer. web based collation tools for tei," *RIDE*, 2020.

[3] A. C. Williams, A. Santarsiero, C. Meccariello, G. Verhasselt, H. D. Carroll, J. F. Wallin, D. Obbink, and J. H. Brusuelas, "Proteus: A platform for born digital critical editions of literary and subliterary papyri," in *2015 Digital Heritage*, vol. 2, 2015, pp. 453–456.

[4] R. Haentjens Dekker, D. van Hulle, G. Middell, V. Neyt, and J. van Zundert, "Computer-supported collation of modern manuscripts: CollateX and the Beckett Digital Manuscript Project," *Digital Scholarship in the Humanities*, vol. 30, no. 3, pp. 452–470, 03 2014. [Online]. Available: https://doi.org/10.1093/llc/fqu007

[5] D. A. Smith, J. A. Rydberg-Cox, and G. R. Crane, "The perseus project: A digital library for the humanities," *Literary and Linguistic Computing*, vol. 15, no. 1, pp. 15–25, 2000.

[6] S. Hagel, "The classical text editor. an attempt to provide for both printed and digital editions," *Digital philology and medieval texts, Pacini, University of Michigan*, pp. 77–84, 2007.

[7] R. R. D. Turco, G. Buomprisco, C. D. Pietro, J. Kenny, R. Masotti, and J. Pugliese, "Edition visualization technology: A simple tool to visualize tei-based digital editions," *Journal of the Text encoding initiative*, no. 8, 2014.

[8] S. Zenzaro, D. Marotta, and A. Bertolacci, "Ceed: a cooperative web-based editor for critical editions," *AIUCD 2018*, p. 93, 2018.

[9] R. H. Dekker and G. Middell, "Computer-supported collation with collatex: managing textual variance in an environment with varying requirements," in *Supporting Digital Humanities 2011: Answering the unaskable*, 2011.

[10] R. Haentjens Dekker, D. Van Hulle, G. Middell, V. Neyt, and J. Van Zundert, "Computer-supported collation of modern manuscripts: Collatex and the beckett digital manuscript project," *Digital Scholarship in the Humanities*, vol. 30, no. 3, pp. 452–470, 2015.

[11] L. J. Sampsel, "Voyant tools," *Music Reference Services Quarterly*, vol. 21, no. 3, pp. 153–157, 2018.

[12] L. Burnard, *What is the Text Encoding Initiative?: How to add intelligent markup to digital resources*. OpenEdition Press, 2014, no. 3.

[13] W. AlKendi, F. Gechter, L. Heyberger, and C. Guyeux, "Advancements and challenges in handwritten text recognition: A comprehensive survey," *Journal of Imaging*, vol. 10, no. 1, 2024. [Online]. Available: https://www.mdpi.com/2313-433X/10/1/18

[14] A. M. Del Grosso and O. Nahli, "Towards a flexible open-source software library for multi-layered scholarly textual studies: An arabic case study dealing with semi-automatic language processing," in *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014, pp. 285–290.

[15] H. Jahanshahi, M. Cevik, J. Navas-Sú, A. Başar, and A. González-Torres, "Wayback Machine: A tool to capture the evolutionary behavior of the bug reports and their triage process in open-source software systems," *Journal of Systems and Software*, vol. 189, p. 111308, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000565

[16] J. Van Zundert, "If you build it, will we come? large scale digital infrastructures as a dead end for digital humanities," *Historical Social Research/Historische Sozialforschung*, pp. 165–186, 2012.

[17] A. Ciula, Ø. Eide, C. Marras, and P. Sahle, "Models and modelling between digital and humanities. remarks from a multidisciplinary perspective," *Historical Social Research/Historische Sozialforschung*, vol. 43, no. 4, pp. 343–361, 2018.

[18] W. McCarty, *Humanities computing*. Springer, 2005.

[19] D. Buzzetti, "Digital representation and the text model," *New Literary History*, vol. 33, no. 1, pp. 61–88, 2002.

[20] M. Beynon, S. Russ, and W. McCarty, "Human computing—modelling with meaning," *Literary and Linguistic Computing*, vol. 21, no. 2, pp. 141–157, 2006.

[21] T. Orlandi, "Ripartiamo dai diasistemi," *I nuovi orizzonti della filologia, ecdotica, critica testuale, editoria scientifica e mezzi informatici elettronici, convegno internazionale 27-29 maggio 1998. Atti dei convegni lincei*, pp. 87–101, 1998.

[22] P. Monella *et al.*, "Livelli di rappresentazione del testo nell'edizione del de nomine di orso beneventano," *Umanistica Digitale*, vol. 2, pp. 67–91, 2018.

[23] F. S.-E. GIOVANNETTI, "Un modello per domarli tutti: verso una rappresentazione del testo come esplicitazione di documento, lingua e contenuto2," *FARE LINGUISTICA APPLICATA CON LE DIGITAL HUMANITIES*, p. 145, 2022.

[24] L. Skyttner, *General systems theory: ideas & applications*. World Scientific, 2001.

[25] R. L. Ackoff, "Towards a system of systems concepts," *Management science*, vol. 17, no. 11, pp. 661–671, 1971.

[26] E. Börger, "The asm ground model method as a foundation of requirements engineering," *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pp. 145–160, 2003.

[27] E. Börger, *The abstract state machines method for high-level system design and analysis*. Springer, 2010.

[28] E. Börger and A. Raschke, *Modeling companion for software practitioners*. Springer, 2018.

[29] E. Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014.

[30] V. Gervasi, R. Gacitua, M. Rouncefield, P. Sawyer, L. Kof, L. Ma, P. Piwek, A. de Roeck, A. Willis, H. Yang, and B. Nuseibeh, *Unpacking Tacit Knowledge for Requirements Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 23–47. [Online]. Available: https://doi.org/10.1007/978-3-642-34419-0_2

[31] M. Polanyi and A. Sen, *The tacit dimension*. University of Chicago press, 2009.

[32] G. Cacioli, G. Cerretini, C. Di Pietro, S. Maenza, R. R. Del Turco, and S. Zenzaro, "There and back again: what to expect in the next evt version," pp. 212–217, 2022. [Online]. Available: http://amsacta.unibo.it/id/eprint/6848/

[33] E. Domı, B. Pérez, Á. L. Rubio *et al.*, "A systematic review of code generation proposals from state machine specifications," *Information and Software Technology*, vol. 54, no. 10, pp. 1045–1066, 2012.

[34] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of uml models: a systematic review of research and practice," *Software & Systems Modeling*, vol. 18, pp. 2313–2360, 2019.

[35] D. L. Parnas, P. C. Clements, and D. M. Weiss, "Enhancing reusability with information hiding," in *Proceedings of the ITT Workshop on Reusability in Programming*, 1983, pp. 7–9.

[36] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 93–96.

[37] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2019, pp. 1–7.

[38] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 2018, pp. 000 149–000 154.

[39] S. Zenzaro, A. M. Del Grosso, F. Boschetti, and G. Ranocchia, "Verso la definizione di criteri per valutare soluzioni di scholarly editing digitale: il caso d'uso greekschools," p. 20, 2022. [Online]. Available: http://amsacta.unibo.it/id/eprint/6848/

[40] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[41] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[42] Y. Assael, T. Sommerschield, B. Shillingford, M. Bordbar, J. Pavlopoulos, M. Chatzipanagiotou, I. Androutsopoulos, J. Prag, and N. de Freitas, "Restoring and attributing ancient texts using deep neural networks," *Nature*, vol. 603, no. 7900, pp. 280–283, Mar 2022. [Online]. Available: https://doi.org/10.1038/s41586-022-04448-z

[43] R. T. Fielding, "Representational state transfer (rest). chapter 5 in architectural styles and the design of networkbased software architectures," Ph.D. dissertation, Ph. D. Thesis, University of California, Irvine, CA, 2000.

[44] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, and U. Sarid, "Openapi specification v3. 1.0," 2022.

[45] J. Hughes and E. Maler, "Security assertion markup language (saml) v2. 0 technical overview," *OASIS SSTC Working Draft sstc-saml-tech-overview-2.0-draft-08*, vol. 13, p. 12, 2005.

[46] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6749

[47] A. M. Del Grosso, D. Albanesi, E. Giovannetti, and S. Marchi, "Defining the core entities of an environment for textual processing in literary computing." in *DH*, 2016, pp. 771–775.