applied sciences

MDPI

Article

Transpiler-Based Architecture Design Model for Back-End Layers in Software Development

Andrés Bastidas Fuertes [1,*], María Pérez [1] and Jaime Meza [2]

[1] Facultad de Ingeniería en Sistemas, Escuela Politécnica Nacional, Quito 170525, Ecuador; maria.perez@epn.edu.ec
[2] Facultad de Ciencias Informáticas, Universidad Técnica de Manabí, Portoviejo 130105, Ecuador; jaime.meza@utm.edu.ec
* Correspondence: andres.bastidas02@epn.edu.ec or andres.bastidas@smartwork.com.ec

Abstract: The utilization of software architectures and designs is widespread in software development, offering conceptual frameworks to address recurring challenges. A transpiler is a tool that automatically converts source code from one high-level programming language to another, ensuring algorithmic equivalence. This study introduces an innovative software architecture design model that integrates transpilers into the back-end layer, enabling the automatic transformation of business logic and back-end components from a single source code (the coding artifact) into diverse equivalent versions using distinct programming languages (the automatically produced code). This work encompasses both abstract and detailed design aspects, covering the proposal, automated processes, layered design, development environment, nest implementations, and cross-cutting components. In addition, it defines the main target audiences, discusses pros and cons, examines their relationships with prevalent design paradigms, addresses considerations about compatibility and debugging, and emphasizes the pivotal role of the transpiler. An empirical experiment involving the practical application of this model was conducted by implementing a collaborative to-do list application. This paper comprehensively outlines the relevant methodological approach, strategic planning, precise execution, observed outcomes, and insightful reflections while underscoring the the model's pragmatic viability and highlighting its relevance across various software development contexts. Our contribution aims to enrich the field of software architecture design by introducing a new way of designing multi-programming-language software.

Keywords: back-end layers; design model; source-to-source transformations; software architecture; software development; transpiler

Citation: Bastidas Fuertes, A.; Pérez, M.; Meza, J. Transpiler-Based Architecture Design Model for Back-End Layers in Software Development. *Appl. Sci.* **2023**, *13*, 11371. https://doi.org/10.3390/app132011371

Academic Editors: Robertas Damaševičius, Sanjay Misra and Bharti Suri

Received: 19 September 2023
Revised: 5 October 2023
Accepted: 8 October 2023
Published: 17 October 2023

1. Introduction

Software architectural design is an essential aspect of software development. Software architectures provide a high-level view of a system by defining its components, their interactions, and the principles that guide their design and evolution. The architecture of a software system determines its quality attributes, such as compatibility, scalability, reliability, maintainability, and performance. Therefore, choosing the correct architecture for a software system is crucial to its success [1]. Structural models perceive software architectures as a composition of components interconnected by additional aspects such as configuration, style, constraints, and semantics [2]. These models are pivotal for capturing and presenting architectural designs, for which formal languages called Architectural Description Languages (ADLs) are used. ADLs simplify the depiction of system components and connections, often using a graphical syntax that resembles "box and line" representations to specify and link components. The adoption of structural models as design models originates from their capability to comprehensively encompass architectural elements and relationships. This promotes efficient communication and analysis of a system design by stakeholders [3].

Software architecture design models are a collection of reusable abstract solutions to commonly occurring software development concerns. They facilitate the development of software components that are dependable, adaptable, and easy-to-manage. These design structures are not specific to any particular programming language or technology; rather, they provide standard design concepts to address recurring problems in software development. The effectiveness of these designs in addressing software development challenges has led to their increasing popularity in recent years. Consequently, various software design models, design patterns, and architectural styles have emerged, including model-view-controller (MVC) [4], service-oriented architecture (SOA) [5], and microservices [6].

Predefined design models offer several benefits to software developers and architects [7]. First, they provide a common language for developers to communicate with each other. By using shared vocabulary, developers can quickly understand each other's ideas, which leads to faster development and better collaboration. Second, software architecture designs reduce the development time and cost. Because these designs are proven solutions to common problems, developers do not need to reinvent the wheel every time they encounter a problem. Instead, existing solutions can be used to build better software systems quickly. Third, software architecture designs improve software quality. By following these designs, developers can build reliable, sustainable, and scalable software systems. These designs ensure that the resulting software system satisfies these requirements. Software systems are becoming increasingly complex and difficult to manage without the set of guidelines and principles provided by common software architecture designs.

Despite the myriad software designs available, the practice of software design remains intricate and often results in suboptimal solutions [8], underscoring the need for novel designs that are tailored to specific objectives.

In the domain of enterprise transactional software, initiating such a project requires comprehensive considerations around software architecture. Conceptual and logical layers, including user interface implementation, business logic, and database connectors, must be meticulously designed. To achieve project milestones, architects must select various technologies for each layer, spanning programming languages to execution platforms, services, and database engines [9]. These choices profoundly influence the software's development trajectory and behavior in production environments. Altering these decisions after development is often an insurmountable challenge. The selection of a specific brands and tools often inadvertently constrains the exploration of other possibilities in the future. This limitation can impede the flexibility and adaptability that large-scale projects demand in their lifecycle. Ideally, decisions regarding the choice of development tools and the associated runtime contexts should operate independently [10].

While selecting specific technologies is a common practice and long-standing dependencies are seldom questioned, there are cases in which such dependencies prove suboptimal. Such cases require flexibility, allowing for modifications to foundational technologies even during maintenance phases. Addressing these dependencies necessitates a shift in software architectural paradigms, allowing technological decisions and deployment platform choices to be made after the development phase. In a regular context, this could entail redeveloping the entire software in a different programming language.

Additionally, certain scenarios demand multifaceted programming solutions. This is evident in software projects for product builders, government agencies, business associations, open-source projects, and others in which diverse deployment scenarios require the same software to be developed in multiple programming languages in parallel or by combining multiple programming languages in a single solution [11]. In this scenario, the entire software may need to be redeveloped in different programming languages, one for each platform to be supported. While this can enhance adaptability across multiple platforms and collaborative development environments [12–14], it represents a solution that requires a great deal of effort.

The adoption of such a multi-faceted approach primarily stems from the urge to reuse extant code to ensure that the requisite functionalities are met. Moreover, exploit-

ing the strengths of specific programming languages augments the implementation of distinctive features, caters to diverse software quality demands, and elevates overall developmental efficiency.

To address these needs, a multi-programming-language software solution is needed in which identical front-end and back-end systems are developed concurrently in multiple programming languages. However, the effort to have the same software developed in several programming languages is very expensive and error-prone [15]. This multi-programming-language (MPL) paradigm is gaining traction, especially in the wake of recent technological advancements [16].

Therefore, a novel approach is introduced that allows development teams to utilize a single unified and transformable programming language. This provide developers with the opportunity to write the software once and subsequently convert it to a different programming language automatically, even during production or maintenance stages, with minimal effort, thereby delivering parallel versions of the same software made with different languages.

Contemporary works in software engineering have led to the advent of transpilers. A transpiler is a tool designed to automatically transform source code made up of a source high-level programming language into another source code made up of a different target high-level programming language, which should be algorithmically equivalent [17].

Beyond their myriad other applications [18], transpilers have found success in front-end web development, empowering developers to employ languages such as TypeScript [19], then subsequently transpile to JavaScript for browser compatibility or even to native mobile platform code. With algorithmic equivalence between source and target languages, transpilers offer a promising avenue for multi-programming-language development, permitting language transformations at any juncture.

Although front-end technologies such as HTML, CSS, and JavaScript are inherently multi-platform-compatible, a comparable solution for the back-end layers remains elusive.

While transpilers primarily handle syntax translation, ensuring that the resulting artifacts run consistently across diverse technologies and platforms requires a more comprehensive approach than merely translating the code. This demands a robust architectural design encompassing the introduction of a development framework, platform-specific artifacts, auxiliary source code generators, and specialized layer designs, among other components. Thus, this research focuses on the role of transpilers as the pivotal element in a novel software architecture design model specifically for back-end layers. In this study, we propose a method to centralize the coding process within a new architectural paradigm for transactional software with the aim of automatically producing multiple implementations of identical software in various programming languages suitable for diverse deployment scenarios.

An empirical experiment was conducted to handle validation for the proposed design, culminating in the development of full-fledged transactional software that embodies the suggested concepts. These artifacts act as a benchmark for gauging the applicability of the proposal and its initial validity. The overarching goal is to pioneer this methodology within emerging software ecosystems, thereby facilitating wide-ranging evaluations in various contexts. This endeavor is poised to collate foundational empirical evidence, bolstering the proposed solution's relevance and suitability for the intended audience.

The objectives for this work are defined as follows:

- Expose sufficient elements about the conforming components of the architecture design model in such a way that they can be consumed by other users as a basis for their own implementations.
- Explain the target audience of this approach in detail.
- Present a comparison with other commonly used architectural designs.
- Identification of future work, especially focusing on ways to increase the external validation of the proposal.

To support the scope of this research, the following research questions were formulated:

- RQ1: What elements should be considered in the software design process when using this new conceptual model incorporating a transpiler as the central development technology for the back-end layer?
- RQ2: Which target scenarios are applicable to software designs that use the new conceptual model that incorporates a transpiler as the core element of the back-end layer?
- RQ3: What are the benefits and challenges associated with implementing a software design model that uses a transpiler in the back-end layer?
- RQ4: How can the effectiveness and validity of the proposed conceptual model be evaluated for software designs using a transpiler in the back-end layer?
- RQ5: How does the proposed conceptual model compare with other software architecture design models?

While we use the term "transpiler" in this paper, there are multiple other terms associated with the same concept that all essentially mean the same thing, including transpiler, transcompiler, source-to-source compiler, s2s compiler, and cross-compiler. Throughout the various sections of this paper, the term "translation" is used to refer to syntax transformations. However, it should be noted that this term is not formally considered a synonym by itself, as it can be ambiguous and confused with other non-computer science fields such as linguistics and education [17].

Although automatic code generation techniques, template-based code generation, domain-specific languages, and even software generation based on artificial intelligence methods may produce multiple programming language outputs, in this article we only cover the application of a transpiler as the core component of our proposal for a new software architecture design model. Future work could include or combine the application of these techniques in order to refine the presented design model.

This study did not involve the creation of a new transpiler or look deeply into the source code translation process. Instead, the focus was on defining design elements to determine the benefit of having a transpiler inside the coding process and the execution pipeline of a transactional application.

As described in this document, the term multi-programming-language software refers to the practice of developing software using multiple programming languages simultaneously during the software development process.

The remainder of this paper is organized as follows. The Section 1 presents an introduction that details the relevant background, problems, objectives, and research questions. Section 2 presents the State of Art and a literature review related to the proposal. Section 3 focuses on the software architecture design proposal, and target usage scenarios. Section 4 presents an empirical experiment that puts the proposal into practice. Section 5 presents a discussion, and Section 6 presents the conclusions and future work.

2. State-of-the-Art

Prior to conducting this study, a review of the relevant literature was conducted [20]. This review aimed to confirm whether any other articles have used the same approach or methods related to the proposed design model. In conducting this initial review, the primary goal was to determine whether there have been any previously published articles that utilize transpilers in the design patterns or in the implementation of back-end layers.

To conduct this review, we utilized the scientific databases Scopus, IEEE Xplore, and ACM. The search criteria included the following conceptual query string: ("transpiler" or "transcompiler" or "source to source" or "source-to-source" or "S2S") and ("design pattern" or "back-end"), filtered only to articles published since 2013. The criteria were oriented to identify articles that were directly related to design patterns or back-end implementations using transpilers in order to determine whether there were any works with an equivalent or similar scope or approach.

The obtained results are presented in the following:

- Scopus: 13 articles

- IEEE Xplore: 9 articles
- ACM: 44 articles

A total of 66 articles were obtained from the three scientific databases. Through reading of the titles and abstracts, we specifically sought to identify any works that could be directly related to the objectives of the present work or that implemented another type of transpiler-based design. A statistical summary of the literature review results is shown in Figure 1.

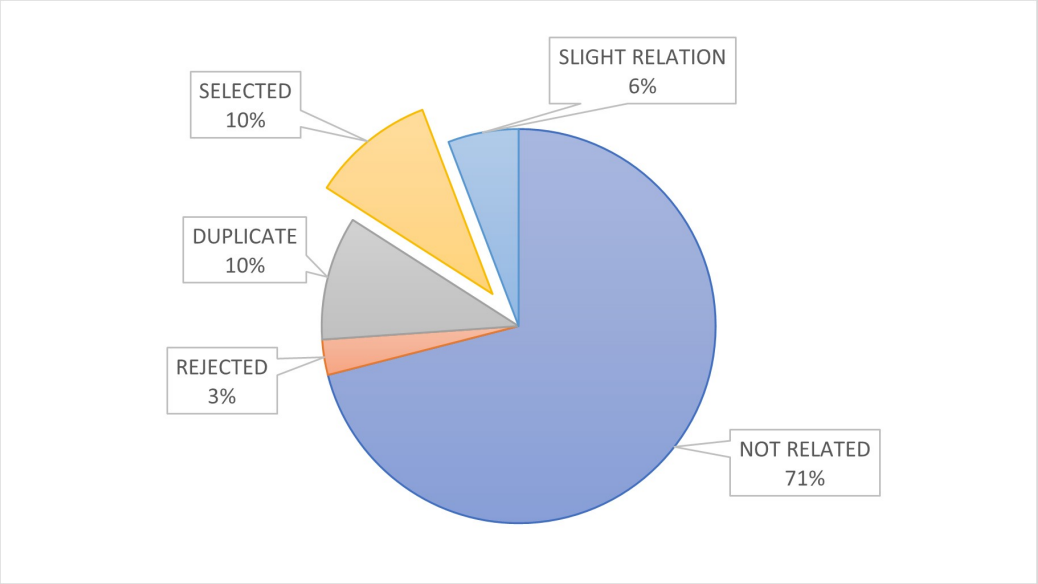

Figure 1. Literature review results.

When we had obtained all relevant articles from the rapid literature review, we carefully evaluated each article to identify those that best aligned with the primary objective of our study. After this literature review, to the best of our knowledge, no other article has followed the same approach and objectives as the current work, opening up the possibility of this being a novel research area, specifically around the proposal of the design model presented in this document.

Several previous authors have highlighted the need for research on multi-language programming. Grichi et al. [21] noted that developers often use multiple programming languages to exploit their strengths and reuse code. However, dependency analysis across multi-language systems is more challenging than across mono-language systems. Vraný et al. [22] suggested that the development of multi-language applications tends to be more expensive in terms of both development and maintenance costs.

Neitsch [23] referenced anecdotal evidence suggesting that single programming languages often do not effectively address the complexities of building multi-language software. Vinoski [24] highlighted that multilingual programmers can leverage the diversity of programming languages to tackle different integration problems, resulting in higher quality solutions that are faster, easier, and less expensive to develop, maintain, and enhance. Mayer et al. [25] noted that programming in multiple programming languages is common in open-source projects.

Transpilers have been used for various types of applications; however, there were no identifiable applications of transpilers for the back-end layers of transactional software [17]. This finding highlights the need for new approaches to facilitate the development of such solutions.

Taken together, the results of our literature review suggest that research on software architecture design in multi-language programming remains necessary and that several complexities exist in the process. In particular, the back-end layer of transactional software presents a unique challenge in the preparation of multi-language software. Addressing this challenge requires the development of new approaches that consider the use of multiple

programming languages along with the inherent complexities of software development and maintenance.

As theoretical and conceptual references for software architectures and design patterns, see Gamma et al. [26], Buschmann et al. [7], Clements et al. [27], Bass et al. [28], Brown et al. [29], Garlan et al. [30], Kruchten [31], Alti et al. [32,33], and Fowler [34].

3. Transpiler-Based Design Model for Back-End Layers

3.1. Design Model

A software architecture design model is a structured and conceptual representation of the organization and function of a software system. It serves as an abstraction that describes the key components of the system, their interactions, design decisions, and the constraints that guide software construction. The software architecture design model provides a high-level view of the system and establishes a foundation for software development and implementation. It acts as a guide for developers and software architects, enabling them to understand the system structure, communicate effectively, and make informed decisions throughout the development process [2].

3.2. Design Fundamentals

A design model is a general repeatable solution for commonly occurring problems in software design. It provides a template for solving problems that can be adapted to satisfy specific requirements [3]. The concept of architecture designs was first introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma et al. [26]. The book presents 23 design patterns categorized into three groups: creational, structural, and behavioral patterns. Each pattern describes a problem, its solution, and the consequences of using that solution. Design patterns have become essential tool for software engineers, providing proven and efficient solutions to common design problems.

Al-Hawari [35] presented a comprehensive example of software architecture designs applied to web information systems. Al-Hawari defined architecture design as a universally applicable and reusable approach for effectively addressing the common challenges encountered in software design. The identification of common patterns in design offers several benefits: (1) accelerating feature development by enhancing the understanding of design details; (2) yielding reusable and reliable classes that can enhance software dependability and reduce development costs; (3) fostering code readability and maintainability by adhering to well-documented design blueprints that are comprehensible to all developers; (4) promoting consistency in the behavior and layout of software modules through the adoption of the same design for recurring visual features, thereby enhancing user-friendliness; and (5) facilitating the creation of more flexible, efficient, and robust software modules, provided that the expected outcomes align with the desired software quality attributes.

Software architecture design typically consists of several key elements [36]. First, the design problem, solutions, and consequences are succinctly described in a single line. The name increases the design vocabulary and allows for higher-level abstraction. Second, the problem describes when to apply the design and explains the problem and its context. Third, the solution describes the abstract elements that make up the design along with its relationships, responsibilities, and collaborations, though not a specific implementation. Finally, the consequences are the results and trade-offs of applying the design, which often concern space and time trade-offs that can impact a system's flexibility, extensibility, or portability. By explicitly listing these consequences, developers can understand and evaluate them better in order to make informed design decisions. Thus, the key elements of the proposed method are as follows:

1. Design Model Name: "Transpiler-Based Design Pattern for Back-End Layers"
2. Problem:
   When there are several benefits of implementing the same software in multiple programming languages simultaneously, developing each one separately in different

programming languages is costly and error-prone. The problem addressed by this design is the complexity and overhead of developing and maintaining back-end layers in various programming languages, frameworks, and platforms. Traditional approaches require developers to manually write and maintain separate back-end implementations for each platform, leading to high development costs and reduced productivity. In addition, maintaining consistency and ensuring equivalence across multiple implementations can be challenging.

3. Solution:
   The proposed solution aims to simplify this process by using transpilable programming languages to write a single implementation of the back-end logic which can then be automatically translated into different target programming languages, platforms, and frameworks. This approach streamlines the development process, reduces development costs, and ensures consistency and equivalence across multiple implementations. The design is intended to be a general-purpose schema, not a domain-specific one, and to benefit a range of scenarios, including software product creators, government agencies, software-as-a-service projects, and open-source initiatives.
4. Expected Consequences:
   The proposed design is expected to have advantages and disadvantages depending on the specific requirements and needs of a software project.
   On the one hand, this approach could offer benefits such as increased productivity and a common coding environment, as well as potentially improved performance, compatibility, and scalability. A single software project can produce several equivalent versions of the same software without any significant effort. This could allow the implementation to be changed to other programming languages and platforms at any time in the future, even when software is in the production stage, thereby reducing dependence on base technologies. In addition, it could allow developers to learn and work in a single programming language while producing code that can run natively on different platforms.
   Potential drawbacks include increased complexity of the build and deployment processes. There may be additional costs associated with maintaining and updating the software compared to a single-programming-language solution. The learning curve for a new programming language and the associated technologies related to transpiler-based architectural elements should be considered as well. Depending on the quality of the selected transpiler, it nay be difficult to ensure that the generated code for each programming language always produces equivalent results when running on each native platform. Certain libraries are not compatible with all transpiler-supported target languages, meaning that certain commonly available imports cannot be used or may need to be implemented from scratch. Documentation and coding references can be difficult to find, as occurs with any emerging technology. Debugging solutions could be challenging because of increased technological diversity.
   Therefore, experimentation should seek to refine and confirm the results presented herein. The decision to use this approach should be based on a careful consideration of the specific requirements and constraints of the specific software project under consideration. Overall, as this design can be considered for use in any kind of software project and there are no proposed limits, architects may consider using this design when flexibility and language interoperability are high priorities and when the benefits outweigh the potential costs and drawbacks.

The following subsections describe the internal design and its elements.

3.3. Abstract Design

The proposed design is based on the use of transpilers to generate different layers of software applications automatically. This approach simplifies and accelerates the development process, allowing developers to focus on the application's specific business logic.

This section presents the study design in a simple and abstract manner, highlighting its primary characteristics.

Figure 2 presents the abstract design, including relationships between the internal elements. For better understanding, the following subsections detail the sequential process and the events that need to occur in order to achieve a multi-programming-language software implementation using the proposed design.

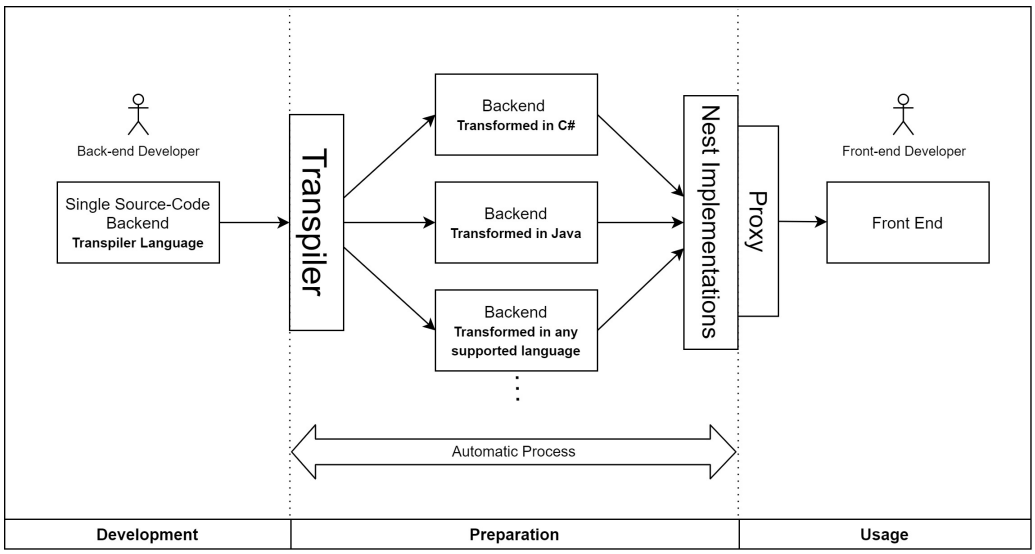

Figure 2. Abstract design.

1. During the development stage, the back-end developer writes the business objects, business logic, and data access layers in the transpiler language, focusing only on functionality and flow logic.
2. During the preparation stage, a transpiler is used to transform the code into different supported programming languages. Code generation, service generator, code injection, and other techniques are used to adapt the transpiler output and obtain the prepared source code. The obtained code is compiled using native compilers depending on the selected programming language.
3. During the usage stage, the obtained compiled artifact should be deployed over the execution services available for each target platform (nest implementations) such that they can be online as a service. The front-end developer uses proxy classes to connect to the deployed implementation and develop user interface interactions with the back-end.

The process is then repeated such that target programming languages and execution technologies are required.

3.4. Detailed Design

In this section, we present the proposed design model for a software architecture based on transpilers in detail. This design is intended to be a guide for software development teams that want to implement a multi-programming-language solution. The design involves the use of a transpiler, allowing for the automatic generation of different language implementations from single-source code. The design is presented in the form of Architectural Description Language (ADL) [3] and is composed of several layers, each with a specific purpose and set of components. By following this design, developers can create applications that are easily maintained and adaptable to future technological changes [32].

Model-Driven Development (MDD) is a cornerstone in modern software engineering. It emphasizes the utilization of abstracted high-level models to drive the entire software development process. Rather than manually coding software from scratch, MDD leverages tools and frameworks that can automatically generate parts of the software

from these abstract models. This approach not only fosters increased productivity and consistency, it enhances the software's agility and adaptability. When integrated into a well-structured architectural framework, MDD can synergize with other methodologies, including transpiler-based approaches, offering a comprehensive solution that addresses both the structural and behavioral facets of software systems [32,33,37]. We considered MDD to be our base conceptual methodology when designing this proposal, as it presents a model over which parts of the software are generated from abstract models.

3.4.1. The Proposal

A new application design is proposed in which a software developer can build the back-end of a business platform or information system using a transpilable programming language with the objective of being the means by which the business logic, methods of connection to the database, business objects, and others are required for the specific solution. After the transformation process, different versions of the source code produced in the target programming languages must be prepared, integrated, and merged with the components of each platform and their execution services. Finally, they are compiled into their own native forms of execution for each language, as in .net to DLLs or Java to JAR or WAR files; these components can be deployed in the chosen application servers as any native application to finally expose it as a web endpoint. With this, it can operate and meet the requests invoked from the front-end. To ease this process, a communication layer is automatically generated as a proxy library, which is developed in the front-end programming language. Each transpiled version should operate in an equivalent manner to ensure that the front-end can point to any of the published endpoints while always expecting the same result.

The objective of the transpiler in this design is to translate an origin source code into another or other destination code. However, making it applicable in the operating environment of a business platform requires several elements to make it usable for a development team. To achieve these objectives, the implementation of certain instruments is proposed to provide greater applicability on the part of the proposed design. Figure 3 presents the design and definition of the instruments needed in each phase to accomplish the expected results.

Below, we provide conceptual details on the different instruments involved.

- The development stage includes the components of the "Development Kit", consisting of following the main instruments, which are detailed later in the study:
  - Common: a library for implementing standard functions that back-end developers use as a unified layer for development.
  - Security: a library that implements standard algorithms for providing developers with methods for encryption/decryption, best practices applicable to the back-end, and others.
  - ORM: a library that allows connecting to databases and enables registering transactions in a simplified way, leveraging developers to only work with a single language most of the time.
  - Runtime services: a library for providing developers with methods needed for configuration, service execution, call flow control, and others.
- The preparation stage includes the components required for the automated process. The following are the main instruments used in this stage, and are detailed later in the paper:
  - Code generators: tools that take the transpiled code and use reflection and other techniques to generate missing layers, service preparation, code injection, and others to prepare the code for the execution-specific platform. They prepare the final code for native compilation and generation of executable artifacts.

  - Native compilers: the native compilers available for each programming language. Automation calls allow these compilers to obtain the required artifacts compatible with nest implementations.
  - Configuration: the tools needed to configure the server-side solution during the preparation and execution phases.
  - Deployment: tools for deploying artifacts produced in runtime technologies.
- The usage stage encompasses the components necessary for the client-side library. Although this proposal primarily targets back-end layers, a front-end layer is integrated into the approach as well. This integration ensures that front-end developers have a dedicated communication layer, which can facilitate their consumption of the methods provided by the back-end. The process predominantly entails the generation of service class code, with an emphasis on the parameters and return values of the disclosed methods. Moreover, methods for invoking services are exposed, enabling front-end developers to utilize them without the need to craft a separate communication layer. Subsequent sections of this document delve deeper into the primary tools employed in this stage:
  - Proxy classes: automatic code generation in the front-end language, used only for those business objects that are used in business logic methods as parameters or outputs to enable front-end developers to easily call business logic methods.
  - Configuration: the tools needed for the client-side configuration to establish the correct parameters for calling the back-end.
  - Security: a front-end library that implements standard algorithms to provide front-end developers with their usage and compatibility when calling back-end methods.
  - Helpers: a front-end library proposed by back-end developers, with the same processing methods on both sides.

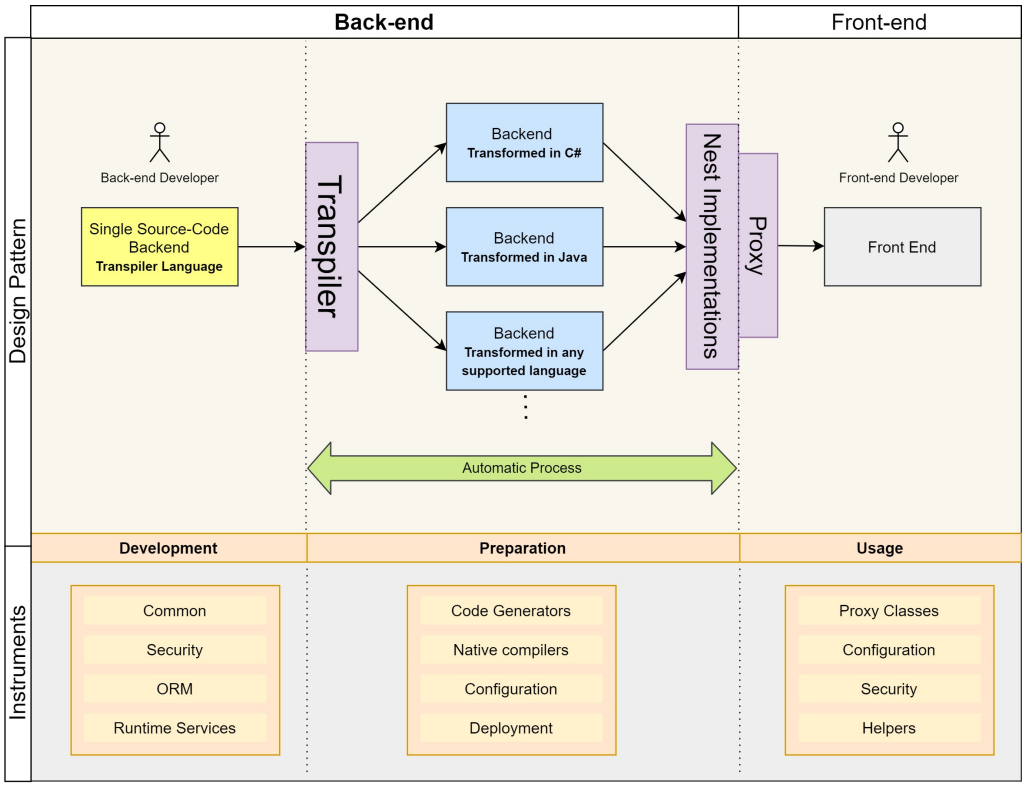

Figure 3. Design instrumentation.

3.4.2. Architecture Design

In this section, all the elements that constitute the proposed design are presented in detail as an autonomous and generic design in such a way that it serves as a high-level functional description that relates to the operating principle of each component, allowing it to be replicated in other scenarios depending on the implementation decisions and technology selection.

The architectural model we propose is outlined in Figure 4; it is based on a layered design conceptualized both as a monolithic application and within the framework of an SOA (Service-Oriented Architecture). This structured approach clarifies our proposal, meticulously outlining the essential components required for its implementation and clearly demarcating the responsibilities and roles of each component while depicting their interconnections. At this initial stage, in order to further emphasize the core concept of transpilation within an application we have opted not to integrate a microservice architecture into the design concept. However, it is worth noting that the primary components are designed envisioning potential compatibility with such an architecture [38], even if our current focus does not underscore this aspect.

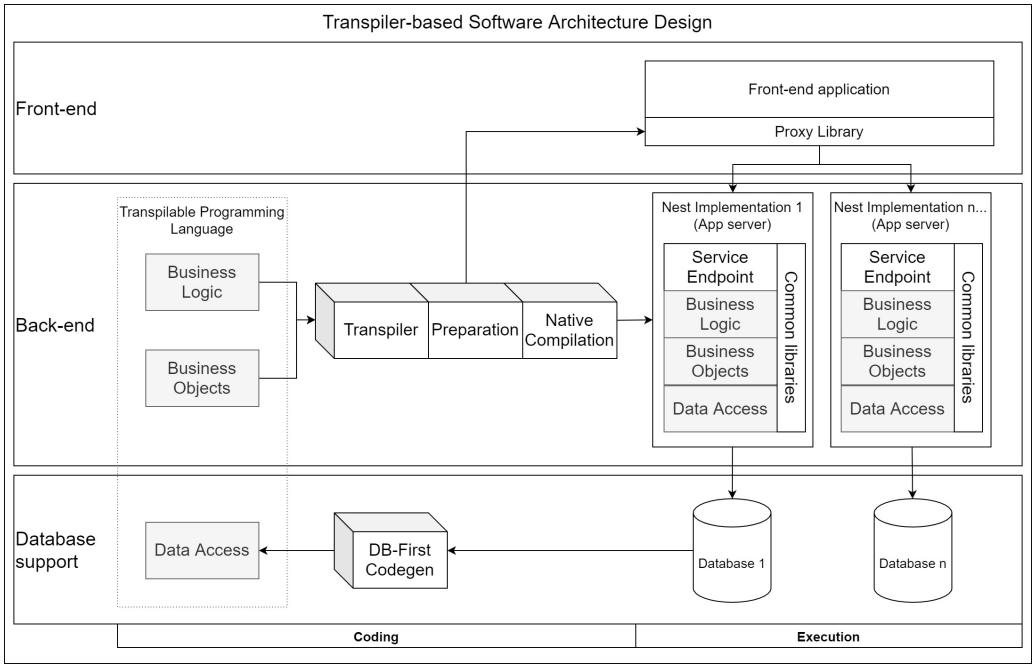

Figure 4. Transpiled-based software architecture design.

Referring to Figure 5, the layouts of the layers and components are as follows:

- Development environment
- Data layer
- Business layer
- Nest implementations
- Presentation layer
- Transversal components

The following sections present the technical specifications of each of these layers.

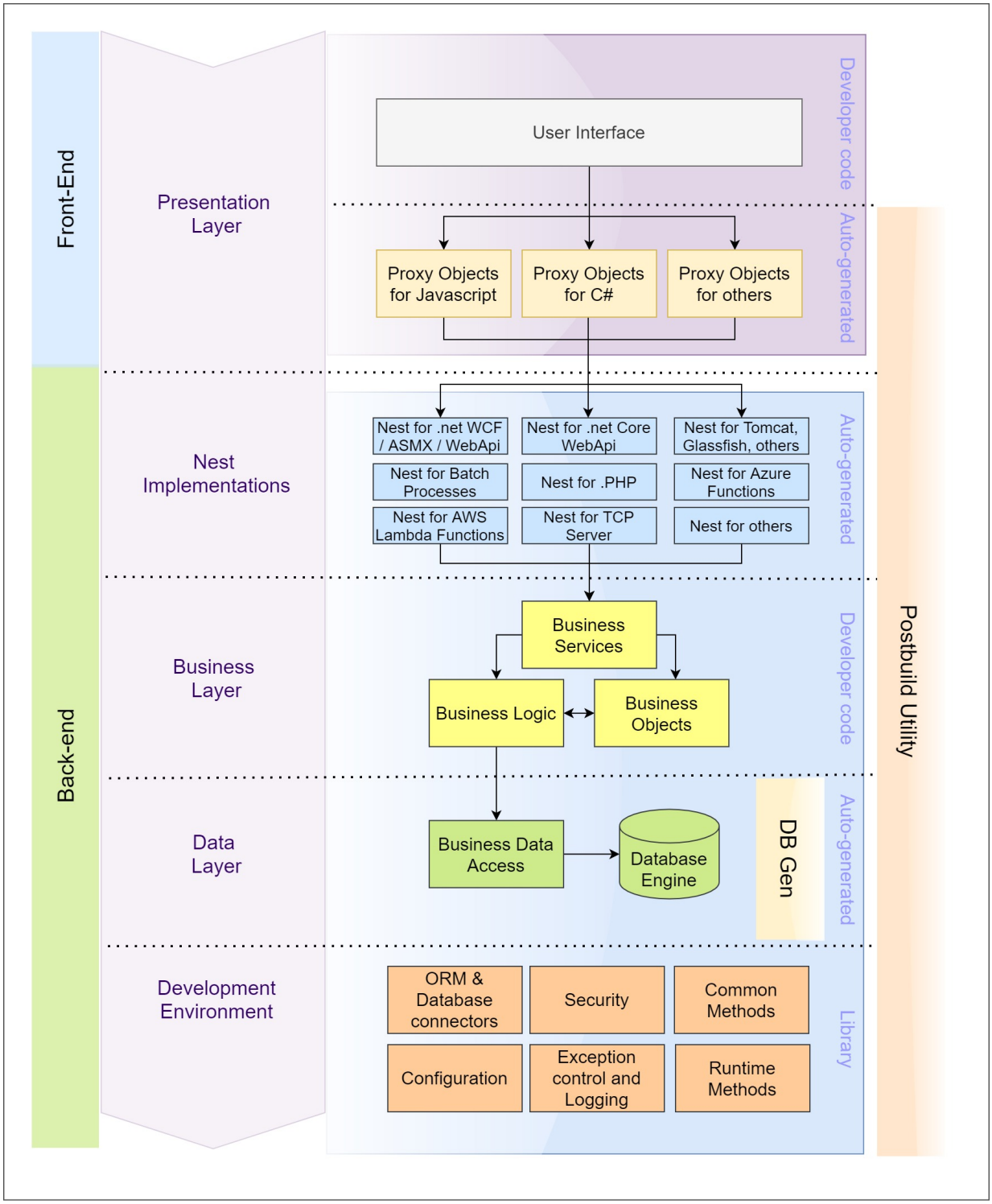

Figure 5. Layered design.

3.4.3. Development Kit

In order for a developer to build the back-end layer using the compiler language, it is necessary to have a base library, as described previously, as instruments are required during the development stage. Therefore, the availability of a development kit that enables the required architectural processes is considered.

This is a programming library that must contain common methods and objects developed in the transpiler language such that all implementations use them as the basis for the development of business logic. Within this layer, there is a definition of common data

types, serialization methods, exception handling methods, methods for handling the application configuration, implementation of an ORM for connection to relational databases, and methods for message processing.

This library can be used for the development of logic and business objects. The developer takes advantage of this programming layer to accelerate the programming process. This library must be considered together with the developed business logic during the transpilation process in order to act as a single set. At the end of the transpilation process, it is expected that both the development framework and business logic will be translated into the target source code, making them compatible for the following stages and for the native compilation of each target technology performed by the following components.

When the solution is already developed, built, and deployed online over the nearest implementations, it is expected that the environment will be compatible throughout the pipeline. This library is responsible for this state of affairs, which is why the nest implementations must consider this library when allowing any transpiled artifact to run.

3.4.4. Automated Process

In Figure 6, it is possible to see the elements of the automated process proposed for each of the operationalization and instrumentation stages of the architecture approach.

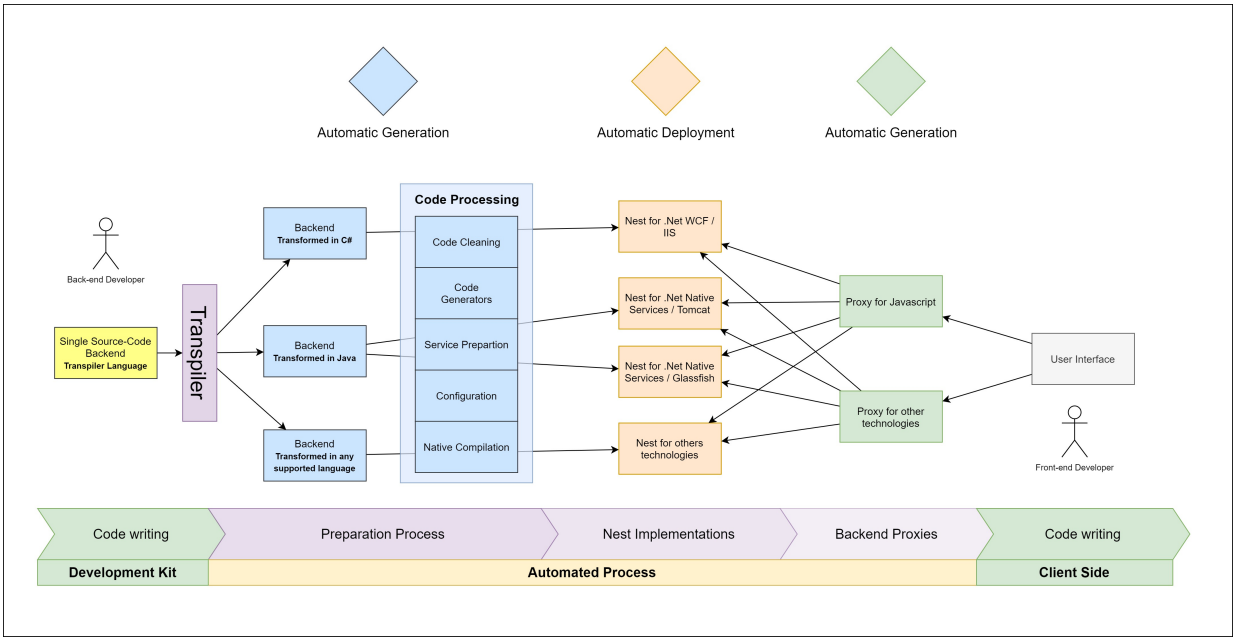

Figure 6. Automated process diagram.

This is designed as a plug-in utility that is executed immediately after the transpiler completes the code translation process. The objective is to prepare the transpiled source code to adapt it to the execution models and compile it in order to make it ready for deployment. This processing is necessary because the source code produced by the transpiler is not directly usable as a standard back-end service, and requires the following three steps:

- Preparation Process
  Starting from the raw transpiled source code, this process executes cleanup processes, splits the code into separate components, generates source code for native service frameworks, prepares for configuration file support, and handles special source code cases and other requirements specific to each supported target programming language. The detailed process is explained in Figure 7. The implementer should consider all post-transpilation processing required to obtain the final code. This code is then automatically compiled by calling the native compilers of each technology to obtain the final artifacts usable on the target platform with all its deployment elements.

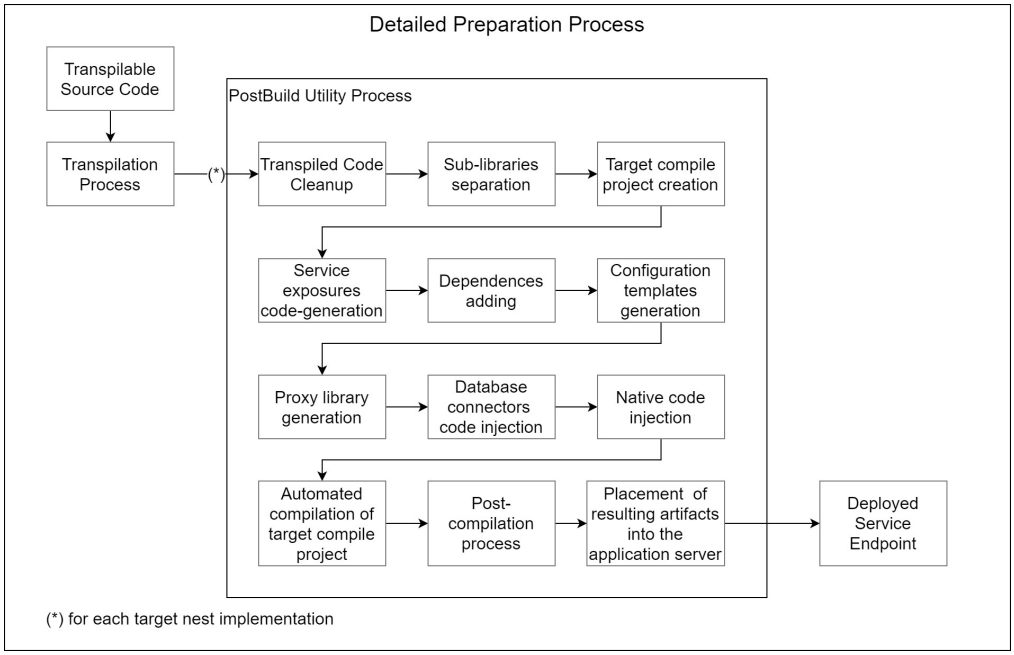

Figure 7. Detailed preparation process.

- Nest Implementation Placement
  The transpiled source code, having passed the preparation process and runtime artifacts, must be placed inside a generic solution that has specific elements of the target technology and supported architecture design. This generic solution is prepared to receive the transpiled code, and is characterized by being compatible with development kit libraries. Finally, the code is automatically moved to the development application server, allowing the developer to begin the testing stage. These native solutions are called "Nest Implementations"; the greater the number of these solutions that exist in the framework, the greater the diversity of ways to generate native artifacts starting from the same business logic. After this stage, the back-end layer is ready for deployment and execution. The final product consists of native artifacts compiled from different programming languages and prepared for different execution technologies.
- Back-End Proxies
  Although the back-end layer of the application was already built in the previous step and is operational, and although the development of the front-end layer is not the main focus of this proposal, it is nonetheless necessary to generate the source code of the service consumption layer. This allows for a reduction in the front-end developer programming time. During this process, business logic objects corresponding to data transport and business logic invocation methods that are exposed should be automatically generated. In addition, this process must incorporate a base library that encapsulates the call methods and support components. A proxy layer must be generated for each supported front-end technology. The generated back-end proxies are considered the "Client Side Library".

3.4.5. Development Environment

To facilitate the development of an enterprise platform or information system using a transpiler, it is necessary to have a library that contains the common elements required by developers, similar to what is available when developing a specific language. These elements must be developed using transpiler language to ensure availability across all supported technologies. Although many elements can be included in such a library, the most important are outlined below:

- ORM and Database Connectors

An enterprise platform or information system cannot exist without the ability to connect to a relational database management system (RDBMS). The provision and storage of data are vital in order for the architecture model to be usable in real implementation scenarios and to permit interaction between the users of the IT solution. Each target platform has its own database connection technology; however, in this case a unified schema is required.

For this purpose, the implementation of native connectors is required for each supported transactional database and target platform in such a way that specific methods are activated according to the development technology and database chosen for the execution of the software.

The use of the SQL language directly within business logic in the form of string concatenation is not recommended, as this factor can generate problems with security (e.g., SQL injection [39]) and maintainability. Therefore, it is suggested that a high-level connection library be provided to connect to the database.

The implementation of ORM (Object–Relational Mapping) is suggested, preferably developed in the transpiler programming language in such a way that the database objects (e.g., tables, views, procedures) can be mapped with their equivalent business objects. This can be used in a simple way in the construction of the business logic, and in the end can be translated into any of the target languages [40]. This approach increases the support of execution technologies by using multiple supported databases. The business logic is database-agnostic when writing the source code. It is not expected that native database connection technologies would be used directly; rather, a single library should serve equally for all technologies.

Database-specific connectors and drivers should be incorporated into the solution using code injection, which occurs during the execution of the post-build utility as part of the preparation stage. These connectors allow for the execution of database sentences directly to the database engines, thereby conforming to the configuration performed at runtime. The connection strings to the databases must be placed in the configuration files using the corresponding configuration components, as is explained later.

For greater independence, support for DDL statements can be incorporated in such a way that the database objects can be auto-generated, permitting support for different modes such as model-first or code-first; this is inspired by the Entity concept in the .net framework [41]. This can be useful for programming stored procedures that can be translated into specific languages of the databases.

- Security

  For any platform built using transpilation, it is important to consider the potential risks posed by third-party attacks. Owing to the technological diversity inherent in such platforms, it is essential to provide cross-cutting methods or mechanisms to mitigate these risks. One approach is to centralize the encryption/decryption methods, authentication management, and authorization of service consumption, as well as input data validators or cleaners for known vulnerabilities, injection mitigation, and similar elements. While this layer is available, the development team is responsible for incorporating these elements into their development planning, particularly with regard to sensitive data exposure validation or access control cases.
- Common Methods

  In software development, various techniques and practices are essential for the development teams. These include serialization methods, compound data types, generic methods that can be employed in business logic, and structured objects that aid information management. The integration of these methods is crucial for the smooth functioning of software and efficient management of data throughout the system.
- Settings

  In order for the platform to operate effectively, a configuration mechanism that can set parameters to initiate its operation is required. Although the connection string to the database is a common parameter, other configuration properties may be required for

IT solutions, such as file directory paths, monitoring and logging schemes, connection parameters with external servers, session-opening parameters in external resources, and global business logic parameters. To address this issue, a uniform configuration mechanism that can be reused across different technologies, such as a group of text-based configuration files, can be placed in a base folder. This folder can be shared through multiple parallel deployments, and the files can be encrypted to prevent unauthorized access or changes.

- Exception Control and Logging
  Exception handling can vary on different platforms. Therefore, it is important to have a unified control and logging scheme for all platforms. To achieve this, exception handling must be implemented in the transpiler language, and common objects for exception control should be defined in this layer. The logging mechanism must record exceptions in detail, including the execution stack, messages, lines of code, or other relevant data. This is particularly useful for analyzing technical events in production or test environments and identifying problems or areas for improvement. To ensure compatibility with different execution platforms, logs should be saved as files in a destination folder, with each file named with unique code for traceability purposes. Informative exceptions should be used for differentiation between business errors and other errors.
- Runtime Methods
  To ensure compatibility and standardization across all platforms, base and attribute classes are necessary for business logic methods, which can then be unified using a single invocation method through external exposure services. Additionally, a router that serves as a gateway for external invocations should be incorporated to manage input and output serialization and dynamically invoke methods centrally with their respective parameters and return objects. This allows nest implementations to customize their invocation based on the input and output flows provided by standard communication while ensuring compatibility with back-end proxies.

3.4.6. Data Layer

In this layer, the developer should implement the data access logic that will later serve to encapsulate the data access mechanisms in the development of the business logic. The considered components are shown below.

Within this layer, developers must implement their data access logic, which is crucial for encapsulating data access mechanisms when building the business logic. The following components are deemed essential for this layer:

- Database Engine
  The proposed architecture aims to provide compatibility with various transactional database engines, allowing users to select the engine that best suits their requirements. These engines store and process information using objects such as tables, views, procedures, functions, and types, which can be utilized in business logic programming. The data access layer is responsible for optimizing the use of these resources, making them readily available in easy-to-use form for programmers.
  As depicted in Figure 8, the database engine's compatibility hinges on the support provided by the nest implementations. Connector classes are to be provided during the preparation stage, allowing the ORM library to effectively process database calls using a corresponding driver that aligns with the specified programming language. The interplay between the connectors and programming languages informs the selection of possible execution technologies, all derived from a single source code. While the RDBMS engines shown in the figure serve as examples to clarify the schema, it is worth noting that any database engine can be incorporated into the nest implementation without design constraints.
  It is crucial to acknowledge that developers might embed business logic directly into databases via stored procedures or custom functions. This can pose a challenge,

as not every element can be effortlessly transferred between database engines. Consequently, it is advisable to confine the utilization of stored procedures to specialized tasks within the database, reducing the effort required during engine migration. As an alternative, ORM can support DDL, allowing it to generate these database logic objects directly in the respective languages of each database all from a singular transpiler implementation.

- Business Data Access
  In this layer, programming objects that correspond to the database objects are added, particularly for the representation of tables and views. The columns represent fields with data types translated into their programming equivalents. The relationships are interpreted as an array of related tables. These objects inherit base classes, have query methods and parameters for query operations, and are linked to the standard ORM query methods. This allows data from the transactional database to be represented in these objects, and their combination can lead to more complex queries.

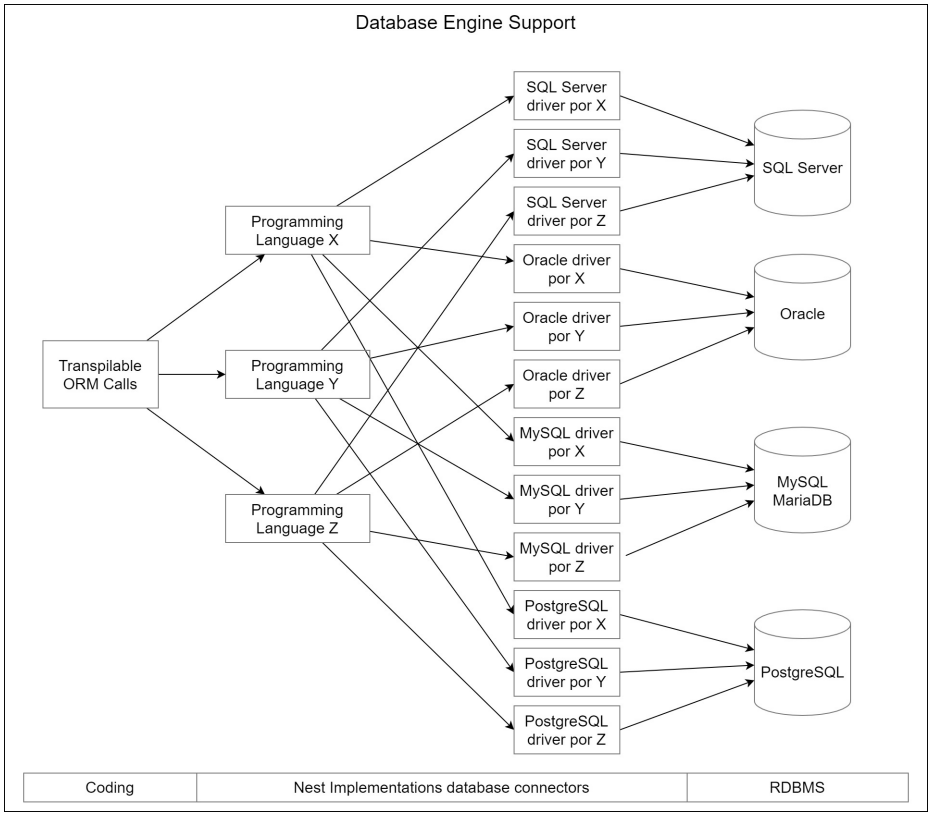

Figure 8. Database engine support.

3.4.7. Business Layer

This layer is dedicated to the implementation of solution-specific methods by the developers. This is the only layer in which developers have direct involvement and can incorporate their source code. The following components are involved in this layer:

- Business Logic
  This layer serves as the core of the architecture, and is where software developers can customize and integrate the necessary algorithms for the operation of an enterprise platform or information system. It is responsible for processing and determining business rules and conditions based on business objects and the data access layer. The implementation of business logic is achieved through methods that are specifically marked to be exposed to the front-end. The development team should primarily focus on this layer; as the other layers are automatically generated, this will lead to reduced development time.

Figure 9 depicts the sequence for exposing a method. The sequence begins with a single transpilable business logic method during the coding stage, which is then transformed into multiple logic method libraries, each in a different programming language. The post-build utilities and preparation processes ready these artifacts for native exposure as well as to facilitate automatic method exposure in various service exposure types, such as SOAP XML web services, API Rest, TCP, or even cloud-based exposures. Ultimately, based on exposure compatibility and technology considerations, the service endpoints can be deployed across different application servers, allowing the developed business logic methods to be invoked as HTTP-based methods. The exposures, programming languages, and application servers shown in the diagram are representative examples to aid comprehension; additional languages, exposure types, or servers can be incorporated without limitations depending on the scope of nest implementation.

- Business Objects
  Business logic methods use input and output parameters. The parameters can be directly taken from the reference to auto-generate objects in the business data access layer. However, in several scenarios it is necessary to generate custom objects that are designed to be a composition of other objects or to extend existing object. These types of custom objects serve to process information in business logic or as a way to transport complex input or output parameters. The development team carries out the component.
- Business Services
  This layer is responsible for automatically generating the method exposure marked in the business logic that is to be included in the nest implementations. It acts as the central point where calls are received, and supports routing and request management throughout their lifecycle. This creates a single connection point for the nest implementation.

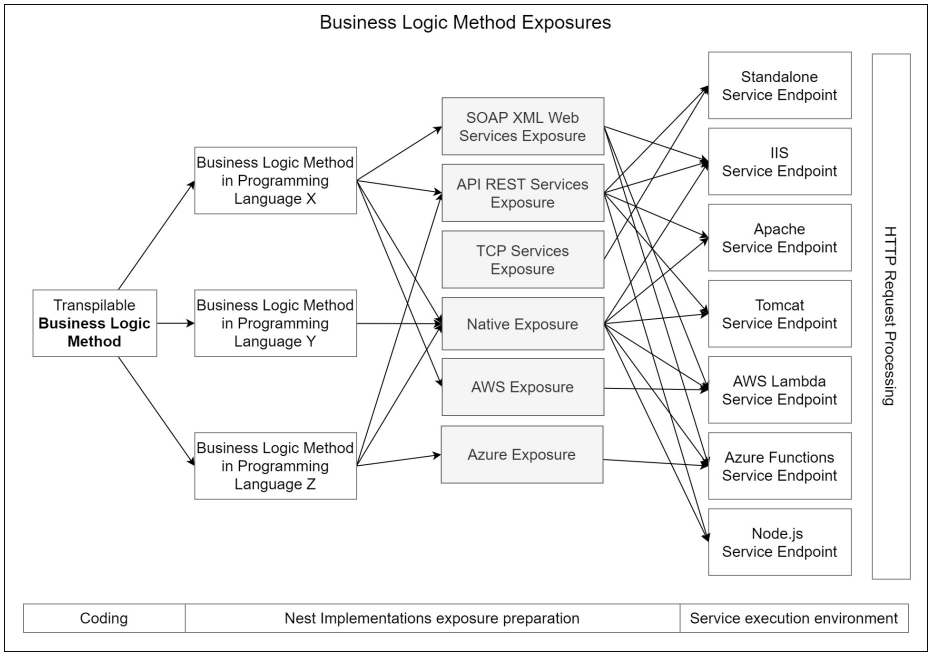

Figure 9. Business logic method exposures.

3.4.8. Nest Implementations

The target technologies may have many different ways of exposing the back-end functionality to the front-end. Each way of exposing these transpiled functionalities through executable artifacts is referred to as a nest implementation.

Multiple nest implementations can be developed to run the same compiled artifacts in different scenarios [42]. For example, the C# language can later expose service methods through XML Web Services, WCF, Remoting, WebApi Controllers, REST Services, and others. If there were a nest implementation for each of these forms, one or several of these technologies could be chosen for deployment at any time without any extra effort being required. Each artifact deployed over a nest implementations has business logic equivalence with the others, even when they are made using different base technologies.

The greater the number of nest implementations incorporated, the greater the capacity of the developed software to support new operating platforms. Even if the transpiler supports a new programming language in the future and particular nest implementations for that technology are incorporated at that time, the existing source code can be recompiled into the new technology directly without major programming efforts.

It might be thought that nest implementation focuses only on the compilation of HTTP communication artifacts or on a particular architectural design; however, it is possible to find them from other natures. For example, nest implementations for the deployment of asynchronous batch processes run business logic methods based on schedules. The remaining implementations may be considered for generating communication via the TCP. A different nest implementation could focus on building and deploying cloud functions such as Azure Functions or AWS Lambda in a serverless schema.

3.4.9. Presentation Layer

The usage stage encompasses components vital for the client-side library, streamlining integration between the front-end and back-end layers. This design allows front-end developers to effortlessly access back-end methods without needing a distinct communication layer. Although the proposed design mainly concentrates on constructing back-end layers, it is necessary to consider certain aspects of the front-end layers in this design model, which are detailed below.

- Proxy Objects
  This layer focuses on the front-end functionality of the architecture. This requires the implementation of an invocation library that enables the same serialization, message packaging, exception control, and service invocation technology as exposed back-end services. In addition, it involves generating invocation methods that correspond to the business logic methods marked for exposure at the front-end. These methods should allow for simple invocation and the use of business objects and data-access objects as input and output parameters. The objects must be developed in the programming language of the user interface, and different built versions of the proxy are necessary to support various front-end technologies.
- User Interface
  This layer is the responsibility of the front-end developers, who implement the presentation and data manipulation logic for user interaction. The developer uses proxy methods to present data and execute transactions, resulting in a fully integrated schema.

3.4.10. Transversal Components

- Post-Build Utility
  The post-build utility plays a crucial role in compiling technology-specific artifacts for each execution environment. It is an essential element that runs across the entire back-end layer and follows the process established in the design proposal. The tool executes the preparation stage in sequence, deploys nest implementations, and generates back-end proxies.
- DB Gen
  If the database-first strategy is employed, wherein the database is designed prior to programming, the DB Gen component should be considered. It can automatically read the structure of tables and views in the database, then generate the programming of this layer in a comprehensive manner that is compatible with the generated ORM.

A DB Gen version is necessary for each database engine brand supported in the generation process.

3.4.11. Behavioural Aspects

Behavioural aspects of the proposal compared with a component view are considered in Figure 10.

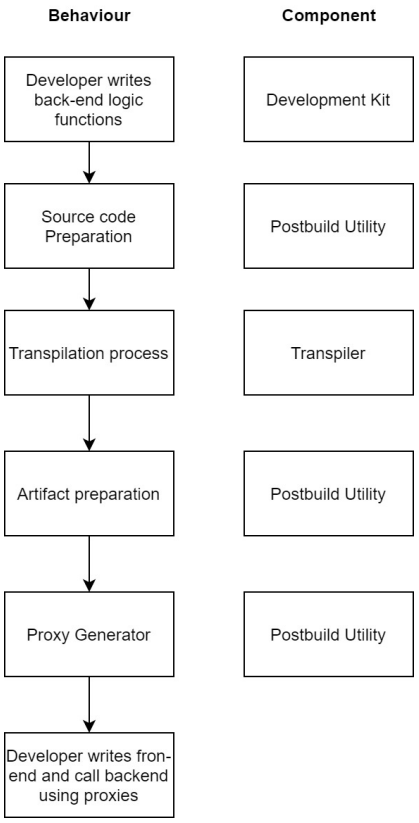

Figure 10. Behavioural aspects vs. components.

3.5. Target Audiences

The proposed architecture design based on transpilers is aimed at software developers and architects seeking a robust and efficient solution for building enterprise applications. The proposed design is particularly relevant for those who work with multiple programming languages and platforms, who must ensure that their applications can seamlessly integrate with different technologies. In addition, it is ideal for teams that need to reduce development times while maintaining a high level of quality and consistency in their code. The possible scenarios that can take advantage of this proposed architectural approach, specifically in the realm of building back-end layers of software, are presented in detail in this section.

3.5.1. Software Product Builders

This refers to companies that create standard wide-ranging software products or platforms aimed at solving common problems. For example, the creators of ERP, BPM, CRM, SCM, DMS, and other software specialize in specific industries (banking core systems, insurance, production, automotive, etc.) designed for multiple clients who use the same solution.

During the design stage of this type of product, a software architect must choose a programming language and the associated technologies. This is an important challenge that seeks to identify the technologies that are most similar to potential customers and the knowledge of the technical team. In this context analysis, analysis subvariables are normally

considered; for example, the platform may be the one with the lowest operating cost for the client, there may be greater support from specialized companies, the learning curve for the development teams and support may be minor, permanent patches and improvements may be necessary, and there may be an active technical community, among many other aspects.

Regardless of the technology chosen by the software architect, there is always a possibility that the chosen platforms are incompatible with what the customer wants or that there is a better option, reducing the chances of product placement or adoption.

It is common that by the time the builder presents their solution to the market or makes a commercial offer, the client has already adopted the technologies that are considered the most convenient for his line of business and investments have been made in licensing, infrastructure, training, contracts with suppliers, support, maintenance, and improvements. Thus, it is probable that the construction base technology of the offered product (operating system, application server, languages, database, and components) will be incompatible with the available technologies of the client.

Even if the software product has the best functional concept and years of valuable experience in the business line, there is a possibility that the client will not finally adopt it because of its incompatibility or the recommendations of the technology or financial managers, as the Total Cost of Ownership (TCO) might have to include new licensing, infrastructure, training, and support, enhancement, and maintenance strategies alongside previously adopted technologies. This could be in addition to the increased complexity of managing technological assets by supporting several technologies simultaneously.

In practice, owing to this restriction, the software producer cannot always adequately reach all customers, which is why it represents the problem of expansion and placement of the product [43]. One possible solution is to invest in developing the product in several programming languages and on several platforms simultaneously in order to present a range of compatible technology options to the client. In this way, each client can choose the one that best suits them. Although this is possible, it is an inefficient and very expensive strategy that multiplies production time and is prone to approval errors, as it is difficult to ensure that all versions work equally in different languages. However, at present there is no viable alternative to consistently address this problem.

3.5.2. Government Software Projects

Owing to the regulations and scope of responsibility and independence of the different entities of the central government and sectional governments, each decides on and incorporates different technologies and base platforms as standards for their operation [44]. They invest in licensing, infrastructure, training, contracts with suppliers, support, maintenance, and improvement.

It must be considered that there are common needs and regulations that any public entity must comply with at its different levels, such as those related to the management of the public budget, compliance with generalized human resources regulations, tax management, control and auditing, land management, and compliance with generally applicable laws and regulations. Although these functionalities are common to all entities, in many cases it is necessary to customize and adapt them to relevant local regulations.

For this reason, it is common for centralized and unified software projects to be proposed and executed by public entities specializing in the provision of software applications, associations, or external providers specializing in the provision or sale of solutions for the government, who seek to develop generic IT solutions that can be used by many public entities simultaneously for the same need in order to prevent each entity from investing on its own in software projects that ultimately, to a large extent, have common requirements and functionalities.

Finally, the solutions intended to be adopted by different entities may not be adopted or used, or may be discarded early. A key factor in this problem is incompatibility with the technologies that institutions already have, which may make it difficult for them to adopt, integrate, and maintain a solution. A second factor is that applying customization

to unique software can break the chain of updating the base platform, giving rise to many different versions of the same product. These elements constitute a barrier to the adoption and maintenance of the proposed generic technologies. No specific methodology provides the possibility of efficiently handling this scenario.

3.5.3. Business Associations

Many companies offer custom software development in consultancy mode. In general, in these cases, a single language and technology are chosen for execution. However, there are certain scenarios in which a consultancy is required to deliver software made in many programming languages simultaneously. This occurs when the contracting party is an association, chamber, or business conglomerate that wishes to jointly finance the development of a specialized solution for its vertical market which is unique, generic, and applicable to all its business members. Because each business member may have its own pre-standardized technologies, the business association prefers to give its partners the ability to select the desired technologies from the built product, and all benefit from a single software development effort. In practice, if this were possible everyone could choose the languages in which they wanted to use the developed software and its source code. At the moment, there is no alternative except to build the software many times with different languages, which is not efficient in either time and cost. Therefore, the most common outcome is that the software is proposed in a single language and companies are required to adapt when this language is not necessarily the most efficient.

3.5.4. Software Projects in Scientific Research

Certain scientific publications or research projects exist, mainly in the fields of engineering, mathematics, artificial intelligence, data science, and similar fields, which, in addition to their scientific contributions published in journals or conferences, include reference software implementations as examples of experimentation or implementation of the proposals presented. Several of these are considered examples of long-lived scientific software [45].

It is normal that the intention of a researcher following the scientific method is that the generated tools can be used for replication of their experiments or for practical implementation of the research results in industry or academic fields.

During the execution of a research project, the researcher must choose a programming language and base platform, which is usually the one closest to their knowledge and experience. The problem is that if the consumers of the resulting software want to use it in another technology, programming language, or platform, or to integrate it into an existing product already built upon other base technologies, there are restrictions that in many cases force them to create long source code transfer processes or forced integration modes.

This results in additional effort, cost, and desynchronization with elements resulting from functional evolution later proposed by the researcher or long-term operation or maintenance problems. The researcher does not have the objective of developing his experimentation instruments multiple times; they only want to use a single programming language and basic technologies. Nonetheless, other researchers or users can make other compatible solutions without much additional effort. Currently, no approach can handle this scenario efficiently.

3.5.5. Creators of Software-as-a-Service

Creators of Software-as-a-Service (SaaS) solutions offer their software on a unique common platform, in many cases paid for by consumption or low-cost monthly payments [46]. This means that customers have no backbone infrastructure to maintain or license. This notably increases the speed of adoption of IT solutions, overcoming basic infrastructure and even compatibility problems. Many companies opt for solutions under this scheme in order to avoid compatibility, licensing, maintenance, support, and evolutionary concerns. This is common for solutions that may have a high level of genericity and require minimal customization. However, unlike the consumer, the story differs for the developer of the

SaaS solution. A critical success factor for the creator of a SaaS solution is to find the integral cost efficiency that makes it possible to offer low prices for each client due to economies of scale without detriment to operating efficiency. This is especially noticeable when dealing with a very crowded multi-tenant solution where there is service provision to many countries or globally and operating costs have begun to grow significantly.

If the software producer initially builds the tool by choosing a single programming language, database, and other base platform elements, this generates a direct dependency on those technologies and their inherent costs, even if these change over time or there is an increase in costs due to vertical or horizontal growth. In this case, the ability to adapt and search for better costs is lesser, being restricted only to searching for alternatives within the same or other cloud providers that handle the initially chosen technologies. This can be limited if it occurs after a few years of platform operation and the technologies evolve. An example of this is when the .net framework is chosen with an Oracle database and it later becomes necessary to run the solution under serverless schemes based on Linux and MariaDB owing to cost efficiency and stability. It would be very expensive to transfer a solution from one technology to another. There is no viable alternative for direct transfer between technologies; however, if available, they would have remarkable independence.

Another factor that comes into play in SaaS-type applications is the desire to make commercial agreements with creators of extensions, integrators, or customizers whose capabilities include working with programming languages other than those used for the development of the application. base platform. In this case, it is common to propose API-type schemes using HTTP calls to ensure interoperability and use of communication standards; however, advanced integrations that require fusion of platform code and extensions to ensure efficiency under high loads are required. The software must run on different platforms simultaneously to take advantage of the code added to particular languages. The software needs to be made in several languages at the same time and executed in parallel on several support platforms. This is an aspect that is difficult to achieve in other scenarios.

3.5.6. Open-Source Projects

In the world of open-source solutions, countless business applications of all types can be adopted by customers without distinction. Thus, it is possible to find solutions such as ERP, CRM, LMS, and CMS, including those specific to certain lines of business [47].

For each open-source project, the project's constitution usually selects the base technologies and platforms on which its development and improvements will work. From that point onwards, there will be a dependency on these technologies. When a solution is functional and becomes popular, there are often projects that port solutions to other languages and technologies so that they work within other execution scopes as well. This is the case for SugarCRM, SplendidCRM, Hibernate, and NHibernate, to name a few examples. Migration projects cannot keep up with the evolution of the original solution, creating gaps in operations, updates, and improvements.

Although the objective of these projects is not the sale of licensing, the benefit for their builders is normally in the service of implementations, customizations, and sometimes in the reputation obtained. Therefore, they seek a solution that is widely adopted in the largest number of possible scenarios. Many companies do not adopt certain open-source software solutions because of the base platform and programming languages with which they were built, as these are different from those that the company has available, even knowing that they are functionally adequate and free. It is preferable to consider solutions that are compatible with previously standardized technologies to move forward. It is of interest both for developers of software projects and for the people or companies that consume them to have a wide range of options for compatible technologies for their use. To obtain a solution that offers this possibility, it would be necessary to program with multiple languages simultaneously, which makes it unfeasible for this type of project.

3.5.7. Long-Time-Use Software

Although software cannot be considered to age in the same way as occurs with physical products, there are parameters by which its possible aging can be measured. One originates from the use of programming languages or old technologies which have not been able to keep up with the evolution of technological tools, which may continue to operate for many years even when the brands themselves have stopped supporting and maintaining them. These systems are often referred to as legacy systems.

As these are stable tools in terms of their development and original business logic, users can handle them with solvency and generally solve the functional aspects required by companies, and there is no strong business incentive to promptly migrate to new technologies. Maintenance, improvement, and evolution costs tend to increase. Companies are motivated to migrate because of the high costs or time that improvements take, or because of incompatibilities that are difficult to overcome when it becomes desirable to integrate with other types of solutions based on new technologies.

It must be considered that when the architecture of such applications was initially proposed the most current technologies available at that time were probably used, and that they finally became legacy systems after a useful lifetime. Therefore, the same can happen today with the selection process of programming languages and base technologies carried out by software architects. There is a chance that certain newly built applications with the best current technologies will within a few years become legacy systems, as well as that this process may become increasingly faster, resulting in an endless cycle. Companies do not see this positively because they feel they should invest in the construction of new software every time there is obsolescence of the base platforms or new technologies, and not necessarily because of the evolution of their business logic when it is stable over time.

There are many scenarios in which companies want their software to be long-lasting in terms of technological validity, stability, and evolution over time without the need to rewrite it every time there are changes in the underlying technology [45]. Currently, there is no method that allows software to exchange its basic operating technologies or programming languages when it has already been built and begun operating.

3.6. Pros and Cons

During the formulation of the proposed design, a number of benefits and disadvantages were identified that should be considered when evaluating its adoption. While this architecture proposes a method to allow the same software to be developed in different languages at the same time, it has several restrictions that could make it unsuitable for different types of projects. In this section, we analyze these in detail.

3.6.1. Pros

- Single Codebase, Multiple Back-End Implementations:
  One advantage of using the proposed design is that it allows for the automatic generation of code for multiple back-end technologies, which can significantly reduce development time and effort. This means that developers can focus on the business logic and front-end layers while the back-end layers are generated automatically using transpilers. This approach can improve the maintainability of the codebase by reducing the amount of manual code that needs to be written and updated.
  The proposed design allows a high degree of flexibility and adaptability during the development process. By separating the layers and automating much of the code generation process, developers can easily make changes and modifications without having to rewrite large portions of the code. In addition, the use of transpilers means that the code can be easily ported to different platforms and technologies, allowing for greater versatility in the final product.
- Single Programming Language:
  Another advantage of using the proposed design is that it allows developers to learn and work with a single programming language throughout the development process,

including the business logic and presentation layers. This can significantly reduce the learning curve for new developers joining the project, and can lead to more efficient development and maintenance of the system over time. In addition, the use of a single language can facilitate communication and collaboration between the different teams and stakeholders involved in the development process.

- Extended Lifetime:
  Programming languages frequently propose new versions of products and services. The same is true for the database and application servers. On the other hand, and under different temporalities, business logic algorithms tend to have different evolution cycles, which can potentially be very fast or very slow depending on the rate of evolution of the functionalities required by companies and their users.
  The evolution of basic technologies may present the effect of obsolescence over time owing to the age of the technology and not to the inapplicability of the algorithms. In certain cases, it is necessary to carry out migration projects from a previous version to a new one by transferring all of the previously developed algorithms solely to take advantage of the new versions of the base technology, for instance due to limitations in the support of the brand, use of related benefits, compatibility with other running products, etc.
  In the proposed architecture design, the business logic is implemented using a transpiler programming language. This makes the layers in which the business logic is implemented agnostic to the temporality of the underlying technology. After time has passed and new versions have advanced, a new nest implementation and its related services can be generated targeting the current technologies at the time, and the solution can be compiled again without the need to rewrite the back-end code. This offers an important benefit to companies and government entities consider software investment as a long-term asset.
- Software Development Methodology:
  From the viewpoint of software development methodology, the proposed design does not establish any key differences from any other software project. It is possible to work with agile or traditional methodologies, as well as with their properties and execution criteria, without any special considerations.
- Reduced Dependency on Base Technologies
  A final advantage of using the proposed design based on transpilers is the reduced dependency on specific base technologies. With this design, the business logic and data access layers are decoupled from the front-end layer and execution technologies, allowing multiple nest implementations that can run on different technologies. This means that the application can be deployed on various platforms without the need for significant changes, reducing the risk of technological obsolescence or a sudden change in politics from a brand, ensuring a longer lifecycle for the software.

3.6.2. Cons

- Focus on Target Audiences:
  Although design formulation starts from the concept of a general-purpose solution, the proposed design may not be suitable for every software project. This architecture is best suited for specific scenarios with well-defined characteristics, and may not be the ideal choice for all projects. Therefore, it is important for development teams to carefully evaluate whether the proposed design aligns with the specific requirements and objectives of their projects before adoption.
- Additional Effort:
  Additional effort and resources may be required to create and maintain the transpiler and its associated components. This could include the development time for the transpiler itself as well as ongoing updates and maintenance to ensure compatibility with new target languages and technologies. Additionally, the need to train developers

in the use of the transpiler and associated tools could add to the overall learning curve and resource requirements.

- Associated Costs:
  Because this approach requires the use of multiple technologies and tools, including the transpiler, nest implementations, and post-build utility, there may be additional costs for training, maintenance, and ongoing support. Additionally, the use of this design may require a development team with specialized skills, which could further increase costs compared to a single-language software solution.
- New Programming Languages:
  The need for developers to learn a new programming language specific to the transpiler could increase the learning curve, and may require additional training resources that add to overall development costs. Additionally, developers who are already familiar with the target language may not be interested in learning a new language, which could make it difficult to implement the proposed design.
- Lack of Documentation:
  Another potential disadvantage of using the proposed design is that it is a relatively new technology, meaning that there may be a lack of documentation and community support. Consequently, developers may face challenges around troubleshooting concerns or finding resources to learn and improve their skills. In addition, a limited number of experts may be available to assist with complex problems, which could lead to delays in development and deployment.
- Difficulty in Debugging:
  Owing to the complex nature of the transpilation process, the multiple technologies involved, and the generation of multiple layers of code, it can be challenging to identify and isolate errors that may arise in the system. This can lead to longer development and testing times, and may require specialized knowledge and tools to effectively debug the system.
- Need for Standardized Libraries: There may be a lack of available libraries in the target languages. Although transpilers can generate code in various programming languages, not all libraries may be available in these languages, leading to limitations in functionality or additional development efforts being required to recreate or find alternative solutions. This can add complexity and potentially increase project costs. Certainly commonly available imports may not be usable, or may need to be implemented from scratch.

3.7. Relationship with Other Architecture Designs

The proposed architecture design model defines the use of a transpiler as a central element of business logic programming. However, this does not contradict what is proposed in other software architecture designs commonly used in the development of enterprise platforms or information systems.

Conceptually, the proposed software architecture design allows the following precepts of component subdivision and specialization to achieve high cohesion and low coupling. Hence, in practice, any design can be used simultaneously, as the conception of the proposal does not specifically determine the design of layers or levels, only the inherent technologies needed for transpilation of the business logic layer and to obtain implementations in multiple target languages.

Because of the way in which the conceptual design and the design of the components have been structured, it is closer to service-oriented architecture (SOA), model-view-controller MVC, or microservices, owing to its characteristic of exposing logic business methods through services.

One of the benefits of writing the business logic layers in the transpiler language is that it decouples the algorithms and processing proposed in the business logic methods from the physical implementation of the architectural layers. This means that the same source code can produce artifacts that conform to another layering scheme or traversal

technology using a different nest implementation that follows another schema without the need to rewrite the core programming.

A detailed analysis of the relationship between the proposed design and other commonly used design patterns is provided below.

3.7.1. SOA

The proposed design based on transpilers can be used in conjunction with the Service-Oriented Architecture (SOA) pattern to enable the development of service-oriented systems. SOA is an architectural style that emphasizes the use of services to support the requirements of business processes. The transpilation process can be used to generate service implementations for each back-end technology, while front-end developers can use the generated proxies to consume services [5].

To apply the proposed design to SOA, the system must be divided into service components that encapsulate the business logic of the system. These components can be deployed in different back-end technologies using the transpiler and nest implementation approaches. The front-end layer can then access these services using the generated proxies, which abstract the differences between back-end technologies.

The use of a transpilation-based architectural pattern can help to improve the interoperability of services by reducing dependence on specific technologies. Together with the proposed design, SOA can help to enable the development of loosely coupled modular systems in which services can be updated and replaced independently.

3.7.2. MVC

The proposed design can be used in conjunction with a Model–View–Controller (MVC) pattern to provide a more robust and scalable architecture. In the MVC pattern, the model represents the application's data and business logic, the view represents the presentation of the data, and the controller handles the user's input and updates the model and view accordingly [4].

Using the proposed design, the back-end layer can be designed to provide a standardized interface for the front-end layer, which can be developed using an MVC pattern. The transpiled code can be used as the model layer, with the front-end layer acting as the view and controller layers. The back-end layer can expose the necessary methods to the front-end layer via nest implementations, allowing the front-end layer to easily access business logic methods.

The use of this combination provides several benefits, including increased scalability, modularity, and separation of concerns. It allows for the development of the front-end layer using a well-established and widely used design while providing a standardized back-end layer that can be easily updated and modified as needed. This approach promotes code reusability and reduces the complexity of the overall architecture, making it easier to maintain and scale the applications.

3.7.3. Microservices

The proposed architecture design based on transpilers can be used in conjunction with microservice patterns. Microservices are software architecture patterns that involve breaking down a large monolithic application into smaller independent services that can be developed, deployed, and scaled independently [6]. The transpiler-based architecture design can facilitate the development of microservices by providing a unified approach to exposing back-end functionality to the front-end layers, regardless of the programming language used in the implementation of each microservice.

In this context, the transpiler can be used to generate the necessary code for each microservice in the programming language of choice and to expose the corresponding API to the front-end layers using a consistent approach, helping to ensure consistency and maintainability across different microservices even if they are implemented using different programming languages. Additionally, the use of nest implementations can allow the

deployment of microservices in different technologies, providing flexibility in terms of supported operating platforms and environments.

However, it is important to note that the use of a transpiler-based architectural design in conjunction with the microservice pattern may add additional complexity to the development process. The decomposition of a monolithic application into smaller microservices requires careful planning and coordination, and the use of multiple programming languages and technologies may require additional training and support. Therefore, it is important to carefully weigh the benefits and drawbacks of using this combination of patterns prior to their implementation in software development projects.

4. Empirical Experiment

This section delves into an initial implementation exercise encapsulated in the design and deployment of a collaborative to-do list application. With its ability to seamlessly function across C#, Java, and PHP from a singular codebase, this application serves as a tangible manifestation of the applicability of the design model. This study represents an initial step into such an architecture; our primary focus is on the validation of the software architecture design model [48]. The methodology, planning, action, observation, and reflection are articulated in the subsequent subsections, laying the groundwork for a comprehensive exploration of this paradigm.

4.1. Methodology

Empirical methodologies offer a robust foundation for understanding and validating innovative paradigms and architectural frameworks in the intricate domain of software engineering. These methodologies, grounded in direct observation or firsthand experience, provide a conduit to gauge concepts beyond mere theory or abstract reasoning [49]. Considering the nuanced intricacies of our transpiler-based architectural design model, a laboratory-based empirical approach is required. Such an environment allows for controlled evaluation, emphasizing technological intricacies over human variables.

For this endeavor, we chose the Action Research Methodology as our investigative backbone, with a distinct emphasis on technology-centric laboratory experiments [50]. While action research traditionally merges action with reflection to foster continuous improvement, our adaptation squarely focuses on the technological facets of our transpiler-based design. By iterating through the stages of reflection, planning, action, and meticulous observation within a controlled laboratory setting, we aimed to deploy and evaluate our to-do-list application while methodically refining the associated architectural model based on the resulting empirical feedback. This methodological approach promises a comprehensive assessment of the technical merits and potential refinements of the proposed architecture.

Figure 11 shows the process following the Action Research Methodology used in this section.

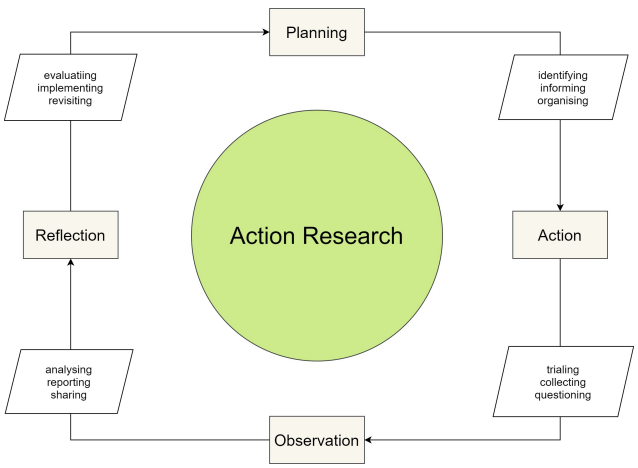

Figure 11. Action Research Methodology process.

4.2. Planning

This subsection delineates the strategic steps and considerations taken to lay the groundwork for our transpiler-based architectural design model experiment.

- Experiment Objective: the primary aim of this experiment was to empirically evaluate the effectiveness of the proposed transpiler-based architectural design model by implementing a to-do-list application as a testbed. Through this application, the experiment sought to assess the model's capability to enable code reusability across diverse platforms, with a specific focus on realizing a unified codebase that can cater to distinct languages such as C#, Java, and PHP. In addition, the experiment aimed to measure the performance, development efficiency, and compatibility of the constructed software artifacts under this design paradigm.
- Tool Selection: in light of this objective, transpilers emerged as the most promising tool, as they allow for the conversion of source code from one programming language to another, potentially bridging the gap between different platforms.
- Transpiler Selection: considering our objective, it was pivotal to choose an effective transpiler to serve as the backbone of our approach. After careful consideration, Haxe (https://www.haxe.org accessed: 12 September 2023, version 4.2.2) was selected because of its flexibility and ability to convert source code between multiple programming languages. This choice aimed to bridge the gap between different platforms, ensuring the feasibility of a unified codebase that could be transposed to languages such as C#, Java, and PHP.
- Implementation Scenario: a controlled laboratory setting was adopted to carry out the experimental evaluation. Such an environment provides the necessary conditions to meticulously assess the the performance of design model while ensuring a focused and unbiased evaluation. The controlled nature of the laboratory setting allowed for replicable conditions, fostering greater credibility and precision in our findings.
- Framework: we utilized the first version of a transpiler-based framework that encapsulates all the concepts outlined in the design model proposed in this paper.
- Model Blueprint: a design model was drafted wherein a singular codebase could be transpiled into different target languages. This model was designed to be scalable, allowing for future additions or modifications based on technological advancements.
- To-do-list Application as a Testbed: for practical validation, a decision was made to design a simple yet comprehensive to-do-list application. This application served as a testbed, allowing for hands-on evaluation of the proposed model's efficiency and robustness.
- Functionality Mapping: functionality was mapped based on the application requirements, including user access controls, task management features, advanced categorization, and dynamic task assignment:
  - User-specific access controls differentiate the views of supervisors and executor users.
  - Task creation capabilities allow both supervisors and executors can initialize tasks.
  - Advanced categorization means that administrators possess the unique ability to manage the available task categories.
  - User management provides an exclusive feature for administrators to regulate system users and their credentials.
  - Comprehensive task details involve recording the task name, detailed description, category, assigned individual, and completion status.
  - Task filtering mechanisms include visual filters based on categories, responsible individuals, and textual searches.
  - Dynamic task assignment grants supervisors the ability to reassign tasks to different executors.
  - Collaborative task visibility provides a centralized system for supervisors to view, modify, and manage tasks across all executors.

- Risk Assessment: potential challenges, such as compatibility issues, code inefficiencies, and transpiler limitations were anticipated. Contingency plans were established to address these risks as the implementation phase began.

4.3. Action

During the action phase of our research, we moved from the theoretical realm of planning into the tangible realm of implementing our transpiler-based architectural design model. Our selected tool for back-end transpilation was the Haxe transpiler; its inherent adaptability and flexibility make it an ideal candidate for executing a uniform codebase across multiple back-end platforms, seamlessly aligning with our research objectives.

A cornerstone of this hands-on phase was the conceptualization and development of the nest implementations. Serving as intermediaries, these components bridge the gap between the source code and the desired back-end platforms. Each nest implementation is crafted with precision and tailored to a specific target technology, guaranteeing faultless integration and operation of the transpiled code in its designated environment. The necessity for distinct nest implementations for each back-end platform underlines the modularity and scalability inherent in our architectural approach, laying a strong foundation for potential future technology integrations.

The project was not confined to the transpiler and nest implementation; in addition, a multi-layered model was established incorporating code generators, automated build programs, and standard libraries specifically designed to interface with databases. This comprehensive setup ensured that every aspect of the back-end system was optimized and ready for interaction with the front-end.

The web application, our empirical touchstone, was assembled meticulously. HTML provided the structural backbone, SCSS lent aesthetic elegance, and TypeScript imbued it with dynamic and type-safe scripting. Together, this trio deliver an interactive, dynamic, and intuitive user interface, further underscoring the potential and practicality of our back-end-focused transpiler model.

With all of these elements harmoniously integrated, we created the to-do-list application. It not only showcased consistent behavior across diverse back-end platforms, it provided robust evidence of the efficiency and versatility conferred by Haxe, our distinct nest implementations approach, and the comprehensive architectural setup underpinning it all.

4.4. Observation

Our journey next led us to a detailed observation period characterized by both quantitative metrics and qualitative insights into the capabilities and performance of our transpiler-based architectural model, with the primary focus on the back-end. The main observation results are as follows:

Holistic Software Realization: one of the most salient outcomes of our observational phase was the creation of complete and functional software. From its foundational layers to its user-facing components, the resulting system was not just a theoretical construct or a fragmented prototype, but a cohesive software artifact. This underscores the viability of our approach; not only can our architectural design and transpiler-based methodology be implemented in practice, it can culminate in a robust end-to-end software solution, as exemplified by our to-do-list application. The Figures 12–14 show examples of the software functionality screens.

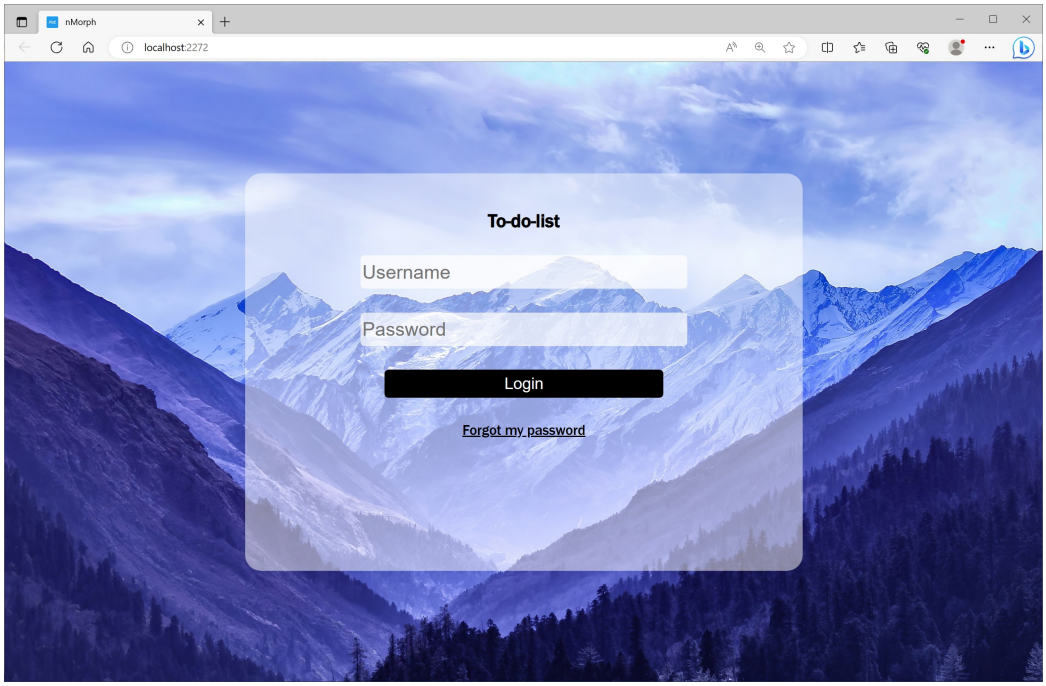

Figure 12. Screenshot of the developed to-do list application (Login Screen).

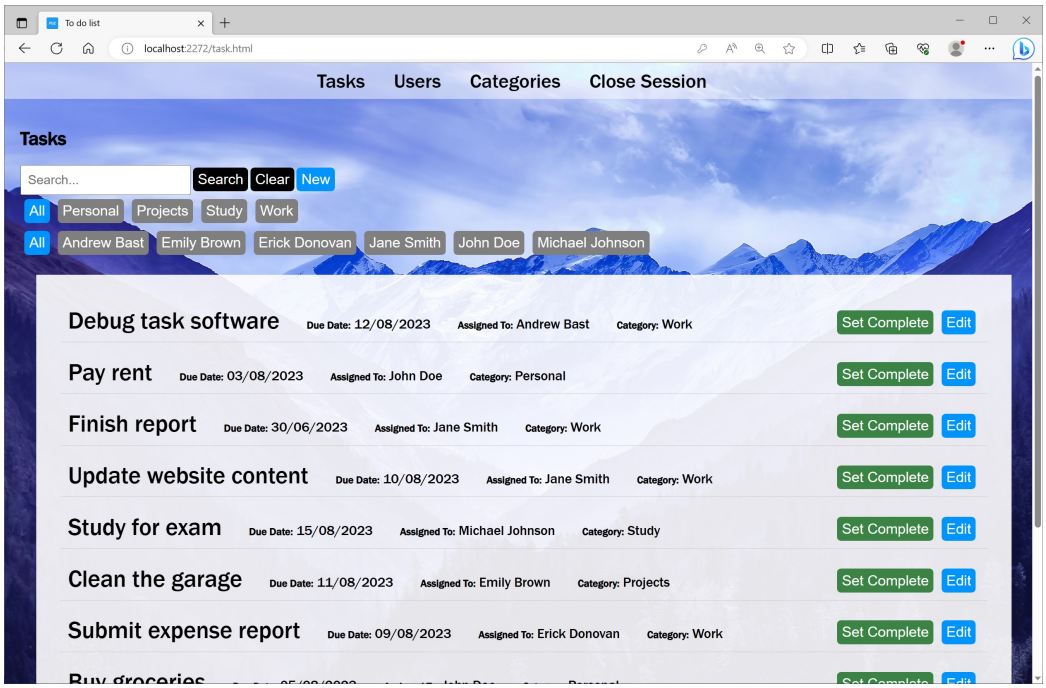

Figure 13. Screenshot of the developed to-do list application (Task List Screen).

Back-end Transpilation via Haxe: Haxe demonstrated remarkable efficacy in back-end source code transpilation. The measurements highlighted variations in execution performance across the targeted back-end platforms with an impressive range of 25–60 ms per service call involving database querying, confirming its efficiency. Figure 15 presents the results of a load test of 100 requests made to the back-end targeting the method that retrieves the task list for the connected user. This was performed on a virtual machine with eight processor threads, 8 gb of RAM, and an SSD unit. The database engine was Microsoft SQL Server version 2019.

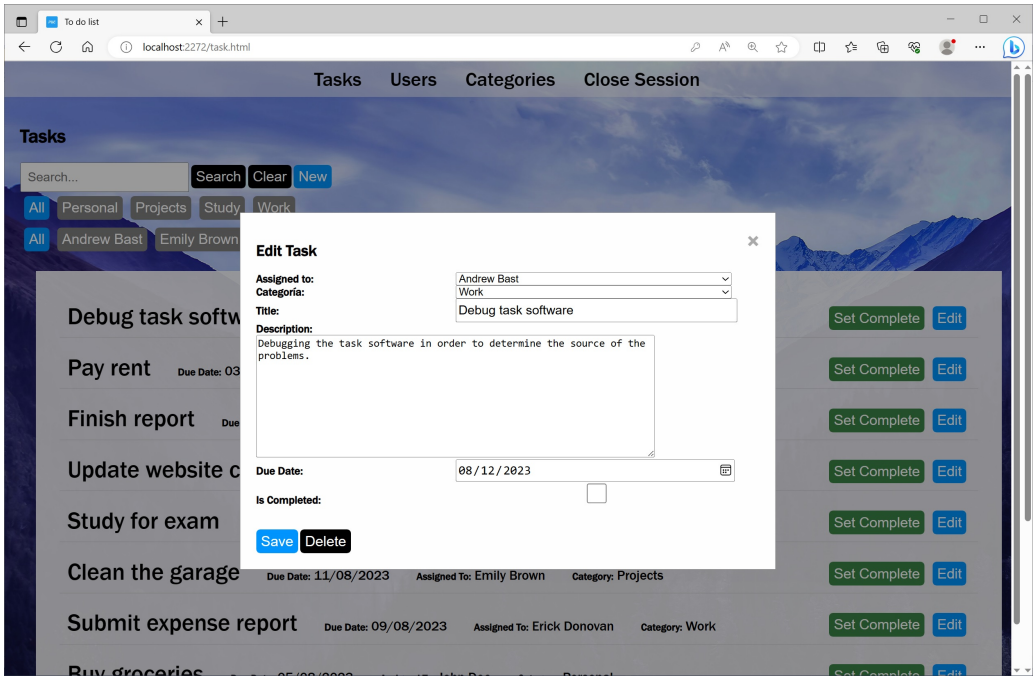

Figure 14. Screenshot of the developed to-do list application (Task Edit Screen).

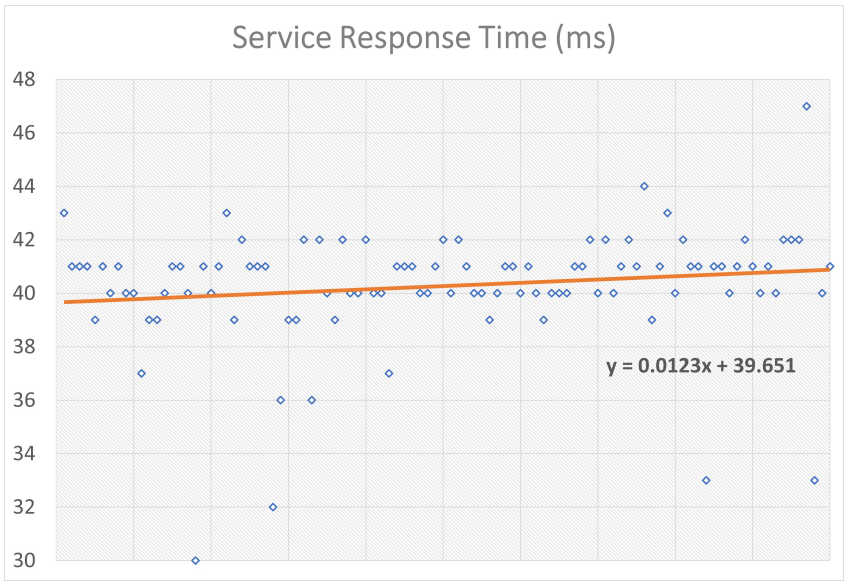

Figure 15. Load test.

Efficiency of the "Nest Implementations": while these components acted as pivotal mediators for our back-end, they also showed compatibility with diverse infrastructural bases, such as C# (.net 4.7.2, .net core 3.1, .net core 5.0), Java (versions 8 and 11), and PHP 7. The seamless integration across these technologies testified to their robust design and flexibility. Figure 16 shows single codebase and transpiled implementations performed using different programming languages. The Task_Search method, which is used in the to-do-list application for listing and filtering the tasks that should be presented depending on the connected user, uses a single ORM implementation to help in querying the transactional database.

Single codebase

```
public function Task_Search(SearchTerm:String, CategoryID: UUID, PersonID : UUID): Array<Task_VTA>
{
    // Dynamic Filter
    var cg = ConditionGroup.SET(Task_VTA.col_Title.LIKE("%" + SearchTerm + "%")
        .OR(Task_VTA.col_Description.LIKE("%" + SearchTerm + "%")))
        .AND(Task_VTA.col_IsCompleted.Equals(false));

    if (!CategoryID.equals(UUID.empty())){
        cg = cg.AND(Task_VTA.col_CategoryId.eq(CategoryID));
    }

    if (!PersonID.equals(UUID.empty())){
        cg = cg.AND(Task_VTA.col_PersonId.eq(PersonID));
    }

    var lst: Array<Task_VTA> = Task_VTA.WHERE(cg).Read(Task_VTA);

    return lst;
}
```

C#

```
public global::Array<object> Task_Search(string SearchTerm, global::nMorph.framework.common.datatypes.UUID CategoryID,
    global::nMorph.framework.common.datatypes.UUID PersonID) {

        global::nMorph.framework.orm.common.conditions.ConditionGroup cg = global::nMorph.framework.orm.common.conditions.ConditionGroup.SET
            (global::nMorph.core.db.sw_todolist.Task_VTA.col_Title.LIKE(global::haxe.lang.Runtime.concat(
            global::haxe.lang.Runtime.concat("%", SearchTerm), "%"))
            .OR(global::nMorph.core.db.sw_todolist.Task_VTA.col_Description.LIKE(global::haxe.lang.Runtime.concat(
            global::haxe.lang.Runtime.concat("%", SearchTerm), "%"))))
            .AND(global::nMorph.core.db.sw_todolist.Task_VTA.col_IsCompleted.Equals(false));

    if ( ! (CategoryID.@equals(global::nMorph.framework.common.datatypes.UUID.empty())) ) {
        cg = cg.AND(global::nMorph.core.db.sw_todolist.Task_VTA.col_CategoryId.eq(CategoryID));
    }

    if ( ! (PersonID.@equals(global::nMorph.framework.common.datatypes.UUID.empty())) ) {
        cg = cg.AND(global::nMorph.core.db.sw_todolist.Task_VTA.col_PersonId.eq(PersonID));
    }

    global::Array<object> lst = global::nMorph.core.db.sw_todolist.Task_VTA.WHERE(cg).Read<object>(
        ((global::System.Type) (typeof(global::nMorph.core.db.sw_todolist.Task_VTA)) ),
        default(global::nMorph.framework.orm.tools.Context));
    return lst;
}
```

PHP

```
/**
 * @param string $SearchTerm
 * @param UUID $CategoryID
 * @param UUID $PersonID
 *
 * @return Task_VTA[]|\Array_hx
 */
public function Task_Search ($SearchTerm, $CategoryID, $PersonID) {
    #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:34: characters 3-182
    $cg = ConditionGroup::SET(Task_VTA::$col_Title-> LIKE("%" . ($SearchTerm??'null') . "%")
        ->OR(Task_VTA::$col_Description->LIKE("%" . ($SearchTerm??'null') . "%")))
        ->AND(Task_VTA::$col_IsCompleted->Equals(false));

    #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:36: lines 36-38
    if (!$CategoryID->equals(UUID::empty())) {
        #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:37: characters 4-55
        $cg = $cg->AND(Task_VTA::$col_CategoryId->eq($CategoryID));
    }
    #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:40: lines 40-42
    if (!$PersonID->equals(UUID::empty())) {
        #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:41: characters 4-51
        $cg = $cg->AND(Task_VTA::$col_PersonId->eq($PersonID));
    }
    #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:44: characters 3-64
    $lst = Task_VTA::WHERE($cg)->Read(Boot::getClass(Task_VTA::class));
    #2_logic\logic\src/nMorph/core/logic/task/TaskLogic.hx:46: characters 3-13
    return $lst;
}
```

Java

```
public static java.lang.String __rtti;

public final haxe.root.Array<nMorph.core.db.sw_todolist.Task_VTA> Task_Search(java.lang.String SearchTerm,
    nMorph.framework.common.datatypes.UUID CategoryID, nMorph.framework.common.datatypes.UUID PersonID)
{
    //line 34 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
    nMorph.framework.orm.common.conditions.ConditionGroup cg =
        nMorph.framework.orm.common.conditions.ConditionGroup.SET(
        nMorph.core.db.sw_todolist.Task_VTA.col_Title.LIKE(( ( "%" + SearchTerm ) + "%" ))
        .OR(nMorph.core.db.sw_todolist.Task_VTA.col_Description.LIKE(( ( "%" + SearchTerm ) + "%" ))))
        .AND(nMorph.core.db.sw_todolist.Task_VTA.col_IsCompleted.Equals(false));

    //line 36 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
    if ( ! (CategoryID.equals(nMorph.framework.common.datatypes.UUID.empty())) )
    {
        //line 37 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
        cg = cg.AND(nMorph.core.db.sw_todolist.Task_VTA.col_CategoryId.eq(CategoryID));
    }

    //line 40 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
    if ( ! (PersonID.equals(nMorph.framework.common.datatypes.UUID.empty())) )
    {
        //line 41 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
        cg = cg.AND(nMorph.core.db.sw_todolist.Task_VTA.col_PersonId.eq(PersonID));
    }

    //line 44 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
    haxe.root.Array<nMorph.core.db.sw_todolist.Task_VTA> lst =
        ((haxe.root.Array<nMorph.core.db.sw_todolist.Task_VTA>) (((haxe.root.Array)
        (nMorph.core.db.sw_todolist.Task_VTA.WHERE(cg).Read(((java.lang.Class) (nMorph.core.db.sw_todolist.Task_VTA.class)) ),
        ((nMorph.framework.orm.tools.Context) (null)) ))) );
    //line 46 "C:\\TodoList\\nMorph\\2_logic\\logic\\src\\nMorph\\core\\logic\\task\\TaskLogic.hx"
    return lst;
}
```

Figure 16. Transformed code.

Layered Model and Supporting Infrastructure: in addition to the sheer functionality, the efficiency of the development process was notably elevated. Although a precise numerical metric was not established, the integration of code generators allowed development efforts to be predominantly channeled into business logic and interface implementation, markedly reducing potential bottlenecks and time wastage. Figure 17 shows the physical implementation of the proposed design model, which is represented by a source code folder structure divided into layers. The equivalent component of the design model is shown on the right side.

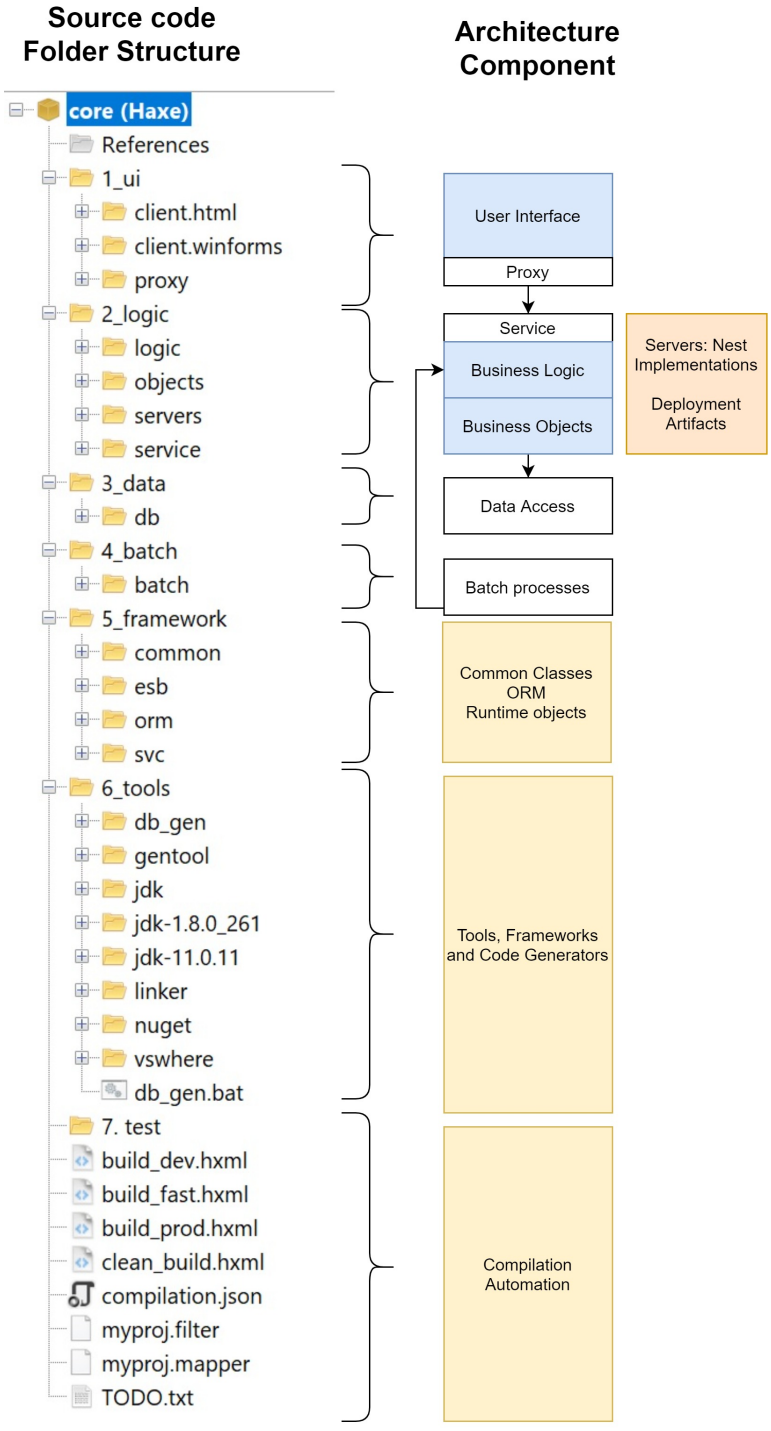

Figure 17. Design model implementation.

Database Connection Objects: the readability and functionality of the objects interfacing with the database were commendable. They were easy to understand and functioned effectively. However, the insights we gathered hinted at potential areas where these could be further refined and optimized.

Interactions with the Front-end: the synergy between the developed front-end (HTML, SCSS, TypeScript) and transpiled back-end was palpable. Furthermore, 100% of the initially outlined functional specification were achieved without any hitches, accentuating the robustness of the employed methodology.

System Interoperability and Scalability: one of the core tenets of this research is the system's capability to function harmoniously across different back-end platforms. We took stock of any discrepancies, mainly technical or performance-related, arising due to varying back-end technologies. The intrinsic modularity of the architecture promises good future scalability and integration with other technologies. In Figure 18, it is possible to find several deployment artifacts generated from a single source code.

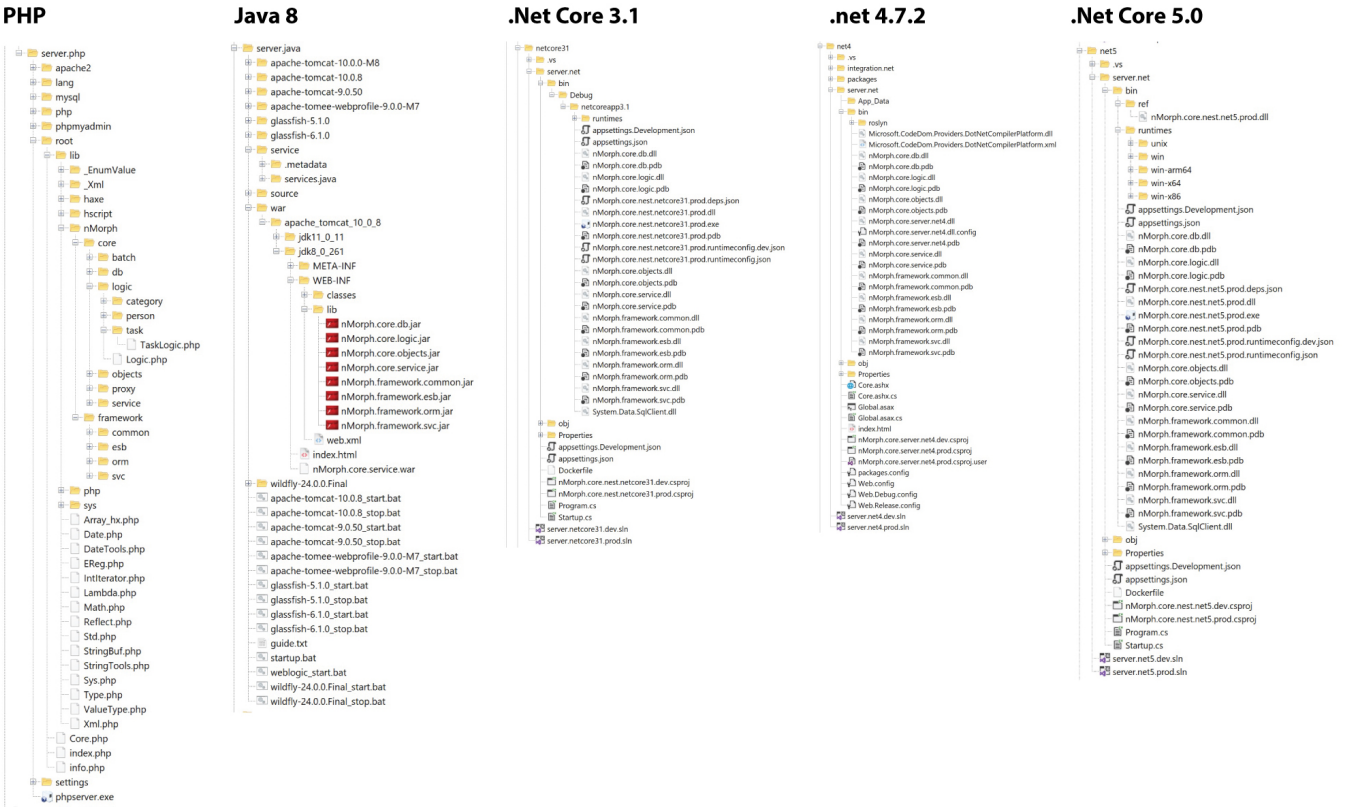

Figure 18. Deployed artifacts.

This observation period, rooted in both tangible metrics and experiential insights, provides a multifaceted view of our application in action, elucidating its strengths and guiding the direction for subsequent iterations and refinements.

4.5. Reflections

Upon culmination of the observation phase, it is indispensable to indulge in a period of introspection and reflection in order to understand the full spectrum of the obtained results and experience. Through the lens of iterative development, the research journey described above presented multiple avenues for improvement and learning.

Comparison with Traditional Approaches: notably, our primary objective was met. In this experiment, the developers wrote the back-end source code just once, yet multiple implementations of the same product were produced, each in a different target programming language. In contrast, traditional approaches would typically yield a solution in

just one technology for the same amount of effort. Furthermore, the time and effort expended by the developers in this experiment were comparable to what would be required when working with a single language, delivering a notable enhancement when using the proposed approach.

Iterative Improvement: one salient realization of this endeavor was the nonlinearity of the development process. The transpiler-based architectural design model did not materialize in its entirety in one swift move; rather, it underwent a series of iterations. Each iterative cycle enriched the source code, refined the layers, and streamlined the processes, ensuring that the architecture matured over time and was responsive to changing requirements and insights. Several future iterations should be considered for future work to ensure that the artifacts can be tested and are reliable.

Challenges in Debugging: one of the most significant hurdles encountered during the experiment was the complexity involved in debugging. Considering the diversity of technologies in addition to the transpilation process, isolating and rectifying issues consumed a disproportionate amount of time, highlighting a critical challenge with this methodology.

Clarifying the Contributions: while the Haxe transpiler played a pivotal role in translating the source code across platforms, the cornerstone of this research and its principal contribution revolves around the architectural design model. The layered design, overall architecture, and paradigm of our approach form the crux of our innovation; while transpilation is the mechanism, architectural wisdom is the message.

Documentation and Framework Limitations: venturing into relatively uncharted territory with Haxe, a less prevalent language, involved its own set of challenges. The scarcity of exhaustive documentation coupled with a limited set of available frameworks occasionally hindered our progress and demanded creative problem solving.

Extendibility and Scalability: while the project showcased a relatively compact functional scope, the foundational structures, methodologies, and layered model ensured its scalability. In essence, the architecture we laid down, while nimble for our immediate needs, is robustly poised to accommodate tools of a significantly higher transactional magnitude.

Tangible Evidence: this venture serves as a evidence of the application of the proposed architectural design model. It is not just a theoretical construct but a practical and tangible entity. Although it successfully demystifies the potential of transpiler-based architectures, its ubiquitous application across diverse scenarios requires further evolution and refinement.

Future Scenarios and Applications: the utility of such an architecture is not confined to its current embodiment; potential scenarios could span the following:

Large-Scale Enterprise Solutions: systems requiring simultaneous operation across different technological platforms can benefit immensely from this design model.

Collaborative Development Environments: a unified codebase can enhance collaboration between teams operating in different technological silos.

Rapid Prototyping across Platforms: the proposed design model could be useful for projects that need to be prototyped or tested across different environments without engaging in redundant development efforts.

Limitations and Opportunities: despite promising outcomes, it must be acknowledged that additional evaluation spaces and diverse use cases would further solidify the validity of the proposed approach. This architectural model, while showing immense promise, remains in its nascent stage. As with all innovations, its widespread adoption across varying contexts will be the true litmus test.

In retrospect, this reflective phase underscores both the successes and the areas in need of enhancement, along with potential future applications and the path forward. As proposed and implemented, this transpiler-based architecture serves as a beacon demonstrating what is feasible today while casting a vision of tomorrow's software development paradigms.

4.6. Replicability

For replication purposes, the source code of the to-do list application presented in this paper has been uploaded to GitHub (uploaded: 18 August 2023) (https://github.com/anfebafu-epn/todolist), including implementations for Haxe, C#, Java, and PHP for the same software.

5. Discussion

5.1. About the Research Questions

This section discusses the theoretical elements of the proposed design model. A review of the research questions posed at the beginning of this study follows:

- RQ1. What elements should be considered in the software design process when using the new conceptual model that incorporates a transpiler as the central development technology for the back-end layer?
  In recent years, transpilers have gained relevance in software development owing to their ability to provide flexibility in the choice of programming languages used in different layers of the application. This proposal presents a new software architecture design that uses transpilers as a central development technology for the back-end layer. However, the question then arises of what elements should be considered in the software design process when using this new conceptual model. In this study, we provide a comprehensive answer to this question. The main elements for consideration are discussed below.
  Layers and Dependencies: one of the main elements to consider in the software design process using this new model is the definition of layers and their dependencies. The back-end layer is composed of the business logic and data access layers, whereas the front-end layer is the presentation layer. In addition, it is important to define the dependencies between the layers and the technologies used to connect them. This includes the nest implementations, which allow different technologies to be used to expose the same business logic methods.
  Code Generation: the code-generation process is another important element to consider. The proposed model uses a post-build utility that compiles the artifacts specific to each execution technology, such as XML Web Services or Windows Communication Foundation (specific services exposures for .net). In addition, a component called DB Gen is used when the database-first strategy is used, which automatically generates the programming of the back-end layer based on the structure of the database. The code generation process must be properly configured to ensure compatibility of the generated code and the transpiled target language.
  Debugging: debugging is an essential part of software development, and the proposed model introduces new challenges. The use of transpilers may complicate the debugging process, as the code being debugged is not the same as that written by the developer. This may require specific tools and techniques for debugging, such as using source maps to map the generated code to the original code. Therefore, it is important to consider the potential impacts on debugging when using this new model.
  Learning Curve: finally, it is essential to consider the learning curve associated with the use of transpilers as a central development technology in the back-end layer. Developers will need to learn new concepts and techniques related to the transpiler and its associated technologies. This may require additional training and resources to ensure successful adoption of the new model. It is important to provide adequate support and resources to facilitate this learning process.
  Summary: the use of a transpiler as a central development technology in the back-end layer introduces new elements to be considered in the software-design process. These include the definition of layers and their dependencies, code generation, debugging, and the learning curve associated with the use of new technologies. It is important to carefully evaluate these elements in order to ensure successful adoption of the

proposed model. By considering these elements, developers can leverage the benefits of transpilers to provide flexibility and efficiency in software development.

- RQ2: Which target scenarios are applicable to software designs that use the new conceptual model that incorporates a transpiler as the core element of the back-end layer?
  The proposed software design incorporates a transpiler as the core element of the back-end layer, and is applicable to a wide range of target scenarios. One of the primary benefits of using this schema is that it reduces dependency on the base technologies used for software development. This makes it ideal for software product builders seeking to deploy software across multiple platforms and technologies. There are several other relevant scenarios in this context, such as government software projects, software for business associations, software projects in scientific research, software-as-a-service projects, open-source projects, and long-time-use software, which can benefit from this software design pattern. A detailed analysis is presented in the section entitled "Target Audiences".
- RQ3: What are the benefits and challenges associated with implementing a software design model that uses a transpiler in the back-end layer?
  For a detailed analysis of the benefits and challenges associated with implementing a software design model that uses a transpiler in the back-end layer, please refer to the section titled "Pros and Cons", which provides a comprehensive overview of the advantages and disadvantages of the proposed software architectural design model.
- RQ4: How can one evaluate the effectiveness and validity of the proposed conceptual model for software design using a transpiler in the back-end layer?
  To evaluate the effectiveness and validity of the proposed conceptual model for software design using a transpiler in the back-end layer, it was necessary to perform a series of tests and evaluations in real-world scenarios. This included testing the performance, scalability, and reliability of the architecture in different contexts as well as comparing it with existing software design models to determine its advantages and disadvantages. Additionally, it was essential to gather feedback from developers and end users in order to assess the usability and overall user experience of the software designed using this model.
  In the most recent phase of our research, we ventured from theoretical planning to the tangible implementation of a transpiler-based architectural design model, for which we used the Haxe transpiler due to its adaptability and flexibility in back-end transpilation. Central to this phase was the development of the nest implementations which acted as intermediaries to bridge the source code with various back-end platforms. In addition, a multi-layered model incorporating code generators, automated build programs, and standard libraries was established to enhance the back-end system, culminating in the creation of a to-do list web application as a practical representation of our design model. This application exhibited consistent behavior across diverse back-end platforms, highlighting the potential of our architectural setup.
  Through our iterative development process, we gained insights into the nonlinearity of creating our architectural model, with each cycle refining and maturing the system. Challenges included complexities in debugging owing to the varied technologies and transpilation processes and the limitations associated with Haxe's documentation and framework availability. Nonetheless, our project showed the applicability of the transpiler-based architectural model, highlighting its scalability and adaptability for future applications. Potential use cases include large-scale enterprise solutions, collaborative development environments, and rapid prototyping across platforms.
  Reflecting on our research journey, the achievements, challenges, and future potential of our transpiler-based architecture have become apparent. This research not only demonstrates the capability of transpiler-based systems, it sets out a vision for future software development paradigms, laying the foundation for potential widespread

adoption in various technological contexts. Further details can be found in the "Empirical Experiment" section of this document.
For future work, a case study should be considered to improve on the validity of our results. In this case, the architecture would be applied to a real-world software development project that would involve the development of a proof-of-concept prototype using the proposed architecture and testing it in a controlled environment. The prototype could then be deployed in a production environment and its performance and reliability monitored and compared with existing solutions. In addition, user feedback could be collected in order to evaluate the usability and user experience of the resulting software.
Another approach could involve benchmarking tests using different software design models, including the proposed model. This would involve the development of similar software applications using different architectures, followed by a comparison of their performance, scalability, and reliability. In this way, the strengths and weaknesses of each model could be identified in order to determine the effectiveness and validity of the proposed model.
It is worth noting that while these approaches can provide valuable insights into the effectiveness and validity of the proposed model, they are beyond the scope of the present work. Therefore, future work is necessary to gradually increase the proposed model's validity. This includes identifying potential limitations and weaknesses of the proposed model and proposing solutions to address them. Nevertheless, the proposed model represents a promising approach to software design with the potential to simplify the development process and improve the quality of software applications.

- RQ5: How does the proposed conceptual model compare with other software architecture design models?
  To answer this research question, it is necessary to refer to the section of the article titled "Relationship with Other Architectural Designs". In this section, a comparison is made between the proposed design and other designs commonly used in software development. This section explains the similarities and differences between the proposed design and other patterns, including MVC, SOA, and microservices, highlighting the advantages and disadvantages of each.
  A comparative analysis was conducted based on multiple articles related to the proposed research with the objective of examining how the proposed design model aligns with or differs from the existing literature in the field of software development. We performed this analysis to enhance the applicability and potential impact of the proposed approach as well as to identify potential areas for further research and development.

5.2. About the Related Articles

A comparative analysis was conducted on multiple articles related to the proposed research with the objective of examining how the proposed design model aligns with or differs from the existing literature in the field of software development. We performed this analysis to enhance the applicability and potential impact of the proposed approach and to identify potential areas for further research and development.

- SequalsK—A Bidirectional Swift–Kotlin-Transpiler [51]
  This scientific article addresses the challenges of developing apps separately for iOS and Android platforms and the pros and cons of using cross-platform development frameworks. The authors suggest a native development approach that takes advantage of the similarities between the Swift and Kotlin programming languages. They propose bidirectional transpiling of the model part of an app, which enables developers to work on a shared model using their preferred language. The proposed SequalsK transpiler supports the critical constructs of both languages and produces correct syntax and semantics. Through a case study, this article shows that the model part of a board game

app can be transpiled in both directions without limitations, with little manual effort required to derive an Android version from a Swift version. The proposed approach is different from using transpilers for back-end development in that it focuses on the model part of the app and enables joint development by iOS and Android experts. This approach does not focus on the back-end.

- FLY: A Domain-Specific Language for Scientific Computing on FaaS [52]
  The authors of this article explored the use of Function-as-a-Service (FaaS) in cloud computing for the development and execution of large-scale scientific computing applications, proposing a domain-specific language named "FLY" for designing, deploying, and executing such applications using the FaaS service model on different cloud infrastructures. This paper presents the design and language definition of FLY on various computing back-ends, and introduces the first FLY source-to-source compiler available on GitHub that supports SMP and AWS back-ends. The FLY language and FaaS service model offer an opportunity for easy development and execution of large-scale scientific applications with fine-grained application decomposition and efficient scheduling on cloud provider infrastructure. The use of FLY is different from a software architecture design that uses transpilers to develop back-end layers, as FLY employs FaaS and a specialized language designed specifically for scientific computing applications.
- OP2: An Active Library Framework for Solving Unstructured Mesh-Based Applications on Multi-Core and Many-Core Architectures [53]
  This article discusses the OP2 library, a framework that allows developers to transform unstructured mesh-based applications into parallel implementations for execution on different hardware platforms. This paper presents the design and features of OP2, including its recent extension to the distributed memory clusters of GPUs, and highlights the main design challenges for parallelizing unstructured mesh-based applications on heterogeneous platforms. The authors used a CFD application written using the OP2 framework for benchmarking and performance analysis on various platforms, demonstrating that OP2 can achieve near-optimal performance without the intervention of domain-application programmers. It should be noted that OP2 is distinct from a software architecture design that uses transpilers for back-end development, which focuses on facilitating the development of back-end layers using transpilers and language interoperability for shared codebase development, whereas OP2 provides source-to-source translation and compilation framework for execution on different hardware platforms and is specifically designed for unstructured mesh-based applications.
- High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs [54]
  This article proposes an automated method to convert programs written in one programming model, such as CUDA, to another model, such as CPU threads, using Polygeist/MLIR, which can achieve performance portability and eliminate the need for manual application portability. This framework allows compiler transformations to be applied seamlessly, including parallelism-specific optimizations. The approach was evaluated by converting and optimizing the CUDA Rodinia benchmark suite for a multicore CPU, resulting in a 58% speedup compared with handwritten OpenMP code. Additionally, this article demonstrates how PyTorch CUDA kernels can run efficiently and scale on a CPU-only supercomputer without user intervention, outperforming PyTorch's native CPU back-end. This approach differs from a software architecture design that uses transpilers for back-end layer development, as it focuses on automatically translating code from one programming model to another rather than developing a shared model that can be transpiled to equivalent versions in different languages.
- Program Transformation Techniques Applied to Languages Used in High-Performance Computing [55]

This article presents an approach to improving the quality of software in high-performance computing (HPC) through meta-programming. Meta-programming allows for the manipulation and generation of programs in order to extend their behavior. The authors introduced the OpenFortran and OpenC frameworks, which enable transparent program transformation for application programmers. In addition, they suggested a generalized meta-programming framework to allow program transformation for any programming language with the aim of solving concerns related to efficiency, scalability, and adaptation in HPC software. This approach differs from a software architecture design that uses transpilers for back-end layer development in that it emphasizes program transformation and generation in HPC rather than language translation.

- Multi-Language Static Code Analysis on the LARA Framework [56]
  This study proposes a method for enhancing source-code analysis to support multiple languages by utilizing the LARA framework to create a language specification for object-oriented programming languages. The authors demonstrated that this approach allows language specifications to be shared between LARA source-to-source compilers and enables the mapping of a virtual abstract syntax tree over the nodes of Abstract Syntax Trees (ASTs) provided by different parsers. To evaluate their approach, the authors implemented a library of eighteen software metrics using the proposed language specification and applied these metrics to the source code in four programming languages (C, C++, Java, and JavaScript), then compared the results to those of other tools. This approach is distinct from software architecture design that uses transpilers for back-end development, as it focuses on abstract syntax tree mapping and analysis rather than automatic language translation and software architecture design patterns.
- Beyond @CloudFunction: Powerful Code Annotations to Capture Serverless Runtime Patterns [57]
  This article delves into the importance of simplicity in the development of elastically scalable applications and addresses how Function-as-a-Service (FaaS), a form of serverless computing, can alleviate this problem. FaaSification frameworks have been introduced to simplify the offloading of code methods as cloud functions; however, as applications become more complex, runtime patterns with background activities require more complex infrastructure orchestration. To address this challenge, the authors of this study proposed a novel approach that uses infrastructure-as-code concepts to simplify orchestration patterns through powerful code annotations. The proposed solution was demonstrated using FaaS Fusion, an annotation library, and a transpiler framework for JavaScript. This approach is distinct from a software architecture design that uses transpilers for the development of back-end layers, as it specifically focuses on simplifying infrastructure orchestration for FaaS applications through powerful code annotations. This enables more efficient and streamlined management of complex runtime patterns. By contrast, a software architecture design pattern that uses transpilers for back-end development can be applied to any type of application, not just FaaS, and may not necessarily address the problem of infrastructure orchestration.

5.3. About the Technical Aspects

5.3.1. Compatibility and Debugging

When working with transpilers, it is challenging to ensure the full compatibility of capabilities across different target programming languages. Consequently, transpilers define their capabilities using their own source language, which guarantees transformability to the target languages. However, there may be specific functionalities that developers wish to incorporate into generic languages. Transpilers often allow the use of external functions to extend the language through references and additional functions and accommodate specific elements of a given programming language.

The challenge with using extended functions is that the equivalent implementation must be manually carried out for each supported target programming language in order to maintain broad compatibility with all languages. This means that when utilizing extended functions developers must meticulously implement the equivalent functionality for each supported target language while ensuring that the platform maintains compatibility across all languages.

By carefully managing the use of extended functions and implementing language-specific counterparts, developers can strike a balance between leveraging the transpiler's capabilities and accommodating the particularities of the individual programming languages. This approach enables the creation of multi-language software systems while ensuring compatibility and functionality across various target languages.

It is important to note that while transpilers provide a powerful tool for code reuse and cross-language development, developers should be aware of the limitations and challenges associated with handling language-specific features while maintaining the overall integrity and effectiveness of the transpilation process.

Furthermore, it is crucial to consider a scenario in which a developer wishes to edit the transpiled code directly. If a developer chooses to make direct modifications, they can achieve a unique and customized level of personalization for their specific target language. However, it is important to note that this approach is not recommended within the proposed design model, as the intention is for the code to be transpiled again to any target language during the maintenance, improvement, and evolution stages.

When the code is transpiled again, the source code is rewritten, potentially eliminating any customizations made specifically for the target language. Therefore, relying on line-by-line transpilation is not practical or functional within this context. Instead, utilizing extension functions is the preferred approach. By leveraging extension functions, developers can extend the capabilities of the transpiler while avoiding the need for manual language-specific modifications. This ensures that the code can be transpiled efficiently and maintains compatibility across multiple target languages.

By emphasizing the use of extension functions and avoiding direct modifications to the transpiled code, the proposed design model aims to strike a balance between customization and maintainability. It enables efficient transpilation and facilitates seamless evolution and updates of software while preserving the broad compatibility offered by the transpiler-based architecture design model.

Another viable alternative is to subdivide back-end programming logic into independent layers using interfaces and invocation schemes. This approach allows for combinations of native and transpiled code. By developing the core of the back-end in the transpiler language and incorporating direct customizations in the target programming language, it is possible to deploy them together on the server without mixing them. Through intercommunication libraries, these components can be made to work together seamlessly.

By adopting this approach, the core of the product can be developed using the proposed design model while allowing for specific customization in separately developed code. This ensures that the code is not overwritten during the back-end maintenance process. Through careful balancing of the transpiled core and customized components, the system can effectively leverage the advantages of the design model while accommodating the unique requirements of each target language.

This approach promotes modularity and flexibility, enabling developers to maintain and enhance the back-end efficiently. By separating the core logic and personalized code, developers can preserve their respective functionalities and facilitate collaboration between transpiled and native components.

Debugging presents several challenges in the context of multi-programming-language software [58]. As the transpilation process involves translating code from a source language to various target languages, debugging becomes more complex due to potential mismatches between the original source code and transpiled code in different languages.

One significant challenge is the potential loss of source-level debugging capabilities when working with transpiled code. Debuggers are often language-specific and rely on source-level information for effective debugging. However, when debugging transpiled code developers may not have access to the same level of source-level information, making it more challenging to accurately track and diagnose issues.

Furthermore, the presence of language-specific constructs and nuances across different target languages can introduce additional complexities into the debugging process. Debugging in one target language may require a different approach or set of tools compared to another, potentially leading to inconsistencies and difficulties in reproducing and resolving bugs.

To overcome these challenges, it is essential to establish robust debugging practices specific to transpilation processes. This may involve implementing language-specific debugging tools or techniques that can bridge the gap between the original source code and transpiled code in different languages. Additionally, thorough testing and validation of the transpiled code in each target language can help to identify and address potential debugging issues early in the development cycle.

By acknowledging and addressing these challenges, developers can mitigate the complexities associated with debugging in a multi-language transpilation scenario, ensuring the stability and reliability of the software throughout its lifecycle.

5.3.2. The Role of the Transpiler

Transpiler technology commonly involves the creation of a specific programming language designed to transform code into other programming languages while adhering to their respective syntax. Consequently, the programming language introduced by the transpiler is considered a high-level language. For example, Haxe, a transpiler, introduces a general-purpose high-level language of the same name capable of translating code into multiple target languages, all of which are high-level themselves. However, it should be noted that the transpiler's language is not limited to high-level languages, and can be translated into low-level languages if supported by the transpiler.

This research project does not propose a new transpiler; rather, it remains open to the use of any transpiler that meets the architectural criteria outlined in the article. Therefore, support for both high-level and low-level programming languages should be viewed as inherent features of the transpiler itself, rather than being specific to the proposed design model.

While the general specification throughout the article may mention C#, Java, and PHP as examples owing to their widespread use, it is important to emphasize that the design model is not exclusively limited to these languages. The proposed design can accommodate various programming languages, both high-level and low-level, depending on the project requirements.

Before a target language of the transpiler can be considered part of the proposed development scheme, whether high-level or low-level, a custom implementation must be developed, particularly focusing on the nest implementation. This entails creating the necessary architectural mechanisms to enable native compilation and execution frameworks that are specific to the target language.

Additionally, the comparison between transpilers and interpreters, although not initially addressed in this article, presents an interesting discussion. Transpilers focus on translating code from a source programming language to a target language, with the resulting code compiled in the target language. In contrast, interpreters run code at runtime, providing dynamic and customizable execution, albeit with potential performance impacts. It is worth noting that there are examples of transpilable libraries that incorporate interpreter-like functionality and blend both concepts.

5.3.3. The Nest Implementation Process

To broaden the proposed solution's compatibility with emerging technologies and fully harness the potential of transpilable code, multiple 'nest' implementations are required. Each of these implementations should be tailored for a specific programming language, targeting a designated application server anticipated to publish endpoints while taking into account the different versions of foundational technologies as well as the inherent compilation tools and database support. The more combinations of these variables offered in distinct nest implementations, the greater the flexibility in selecting the final solution artifacts.

To develop new nest implementations, it is necessary to be well acquainted with the standardized library outputs derived from the transpilation process as well as to possess deep insight into the target technologies. This ensures that the implementer can deliver the core infrastructure for the endpoint, database support, and execution services as well as best practices and any necessary source code generation methods.

In formulating a comprehensive framework that employs this design model, provisions should be made for interfaces and tools that simplify the creation of nest implementations. This is especially pertinent in light of the expected growth in the number of available nest implementations.

5.4. About Previously Published Works

The authors of this article have published previous work on earlier partial advances in the investigation. Below, we describe the relationship of these works to the proposed method:

- Transpiler-Based Architecture for Multi-Platform Web Applications [42]
  This article presented our first published research advance on transpiler-based architecture design, directed at building multi-platform web applications that use a transpiler to write the business logic and service layer for C#, Java, and PHP. An experimental prototype was developed using the Haxe transpiler to test the main idea. Preliminary results were presented.
- A systematic review of Transpiler usage for Transaction-Oriented Applications [18]
  This article discussed the importance of multi-platform and multi-programming-language software architecture design for software product builders from the perspective of increased flexibility, compatibility, and likelihood of adoption. In this article, we presented the results of a systematic literature review conducted to identify current works on transpiler implementations and usage for transaction-oriented applications, concluding that transpilers have not been previously used for designing multi-platform transaction-oriented applications, opening this field up for future research.
- Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry [17]
  This article presented the results of a systematic mapping review conducted to identify the use and implementation of transpilers in research studies over the last ten years. The review identified that the most common uses of transpilers are related to performance optimization, parallel programming, embedded systems, compilers, testing, AI, graphics, and software development. In this article we proposed several future research areas for the use of transpilers, such as transactional software, migration strategies for legacy systems, AI, math, multi-platform games and apps, automatic source code generation, and networking. The goal of this approach was to identify the extent and impact of research sub-areas related to the use of transpilers, rather than to provide a specific solution for a software development problem.

6. Conclusions and Future Work

6.1. Conclusions

A transpiler is a tool that converts source code from one programming language to another while preserving the behavior of the original code. This allows developers to write code in a single programming language and translate it into different target languages.

In this article, we introduce a new software design model that uses a transpiler and its own programming language to write the back-end layers of multi-programming-language software. The design is modular, scalable, and flexible, allowing developers to write the business logic and back-end components only once and then easily transform them into different versions made up of different programming languages. This approach aims to reduce costs, streamline development, and improve the efficiency of multi-programming-language platforms. The design is suitable for several scenarios where it may achieve better results than those that use a single programming technology.

Five research questions were oriented toward the elements needed in a software design model that incorporates a transpiler as the central development technology of the back-end layer, considering its elements, applicability, benefits and challenges, effectiveness and validity, and comparison with other designs.

In the course of this article, sections on design fundamentals, abstract design, detailed design, target audiences, pros and cons, and their relationship with other commonly used designs are presented and detailed.

In the discussion section, the research questions and hypotheses are analyzed and detailed.

Figure 19 shows a generic and simplified sheet presenting the proposed model in abstract form.

In this study, we have detailed new software architecture design model that utilizes transpilers in the back-end layer. This article discusses the main elements to be considered when designing software using this new model, including the layers and dependencies, code generation, debugging, and the learning curve associated with the use of new technologies. By carefully evaluating these elements, developers can leverage the benefits of transpilers to provide greater flexibility and efficiency in software development.

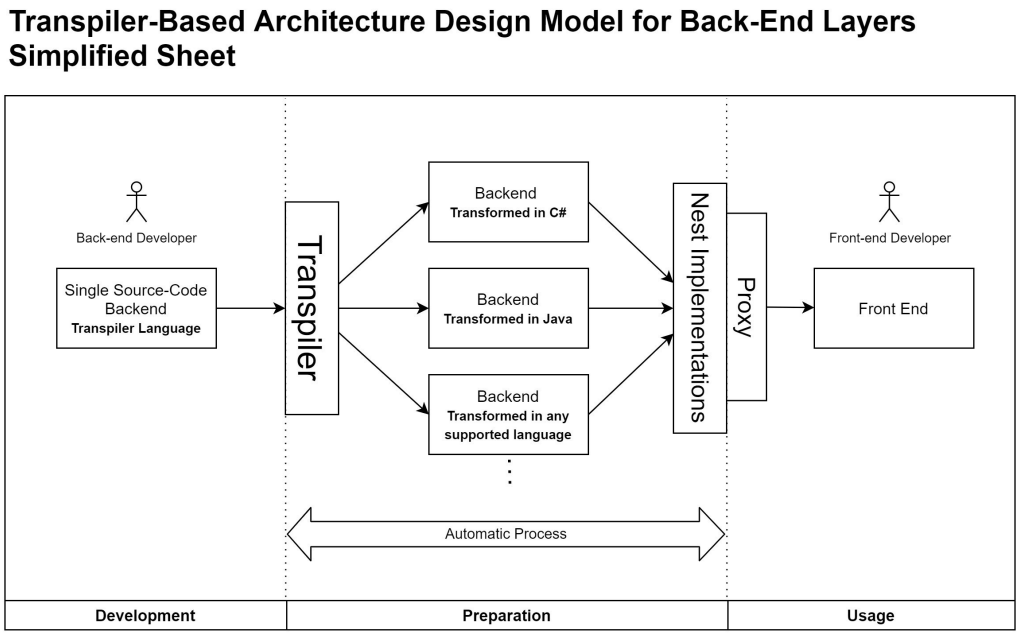

Figure 19. *Cont.*

The Problem

The problem addressed is the complexity and overhead of developing back-end layers of the same software for different programming languages, databases and platforms at the same time.

Traditional approaches require developers to manually write and maintain separate back-end implementations for each platform, leading to high development costs and reduced productivity. Additionally, maintaining consistency and ensuring equivalency across multiple implementations can be challenging.

Also, when there are several benefits of having the same software be implemented in multiple programming languages simultaneously, but developing each one separately in different programming languages is costly and error-prone.

The Solution

The proposed solution aims to simplify this process by using transpilable programming languages to write a single implementation of the back-end logic, which can then be automatically translated into different target programming languages, databases and platforms.

This approach streamlines the development process, aims to reduce development costs, and ensures equivalency across multiple implementations.

The design model is intended to be a general-purpose schema (not a domain-specific one) and mainly beneft to some scenarios, including software product creators, government agencies, SaaS projects, and Open-Source initiatives, among others.

Abstract Design

The proposed design pattern is based on the use of transpilers to generate the different layers of a software application automatically. This approach simplifies and speeds up the development process, allowing developers to focus on the specific business logic of the application. This section will present the design pattern in a simple and abstract way, highlighting its main characteristics.

- During the **Development** stage, the back-end developer writes the business objects, business logic, and data access layers in the transpiler language, focusing only on the functionality and flow logic.
- During the **Preparation** stage, a transpiler is used to transform the code into different supported programming languages. Code generation, services generator, code injection, and other techniques are used to adapt the transpiler output and to get the prepared source code. The obtained code is compiled using native compilers depending on the selected programming language.
- During the **Usage** stage, the obtained compiled artifact should be deployed over the execution services available for each target platform (Nest Implementations), so they can be online as a service. The front-end developer uses proxy classes to connect to the deployed implementation and to develop the user interface interactions with the back-end.

The consequences

The proposed design for multi-programming-language software expects to offer several significant advantages. Firstly, it facilitates the automatic generation of code for multiple back-end technologies, leading to a substantial reduction in development time and effort. Developers can focus on business logic and front-end layers, while the back-end layers are efficiently generated using transpilers, enhancing code maintainability and reducing manual coding. Secondly, the design promotes the use of a single programming language throughout the development process, streamlining the learning curve for new developers and promoting efficient communication and collaboration among different teams and stakeholders. Furthermore, it enables greater flexibility and adaptability during development by separating and automating code generation, facilitating seamless porting of code to various platforms and technologies. The design's ability to support diverse methodologies and its reduced dependency on specific base technologies contribute to its appeal, allowing for efficient deployment and an extended software lifetime.

It also comes with several drawbacks that need to be carefully considered. Firstly, the design's applicability is focused on specific target audiences and well-defined scenarios, making it potentially unsuitable for all software projects. Development teams must conduct thorough evaluations to ensure alignment with their project's specific requirements and objectives before adopting this approach. Secondly, implementing the design requires additional effort and resources, particularly in creating and maintaining the transpiler and its associated components. Training developers on using the transpiler and its tools adds to the learning curve and resource demands. Additionally, there are associated costs for training, maintenance, and ongoing support, especially considering the need for a specialized development team, potentially increasing costs compared to single-language software solutions. The introduction of a new programming language specific to the transpiler further raises the learning curve and development costs, especially for developers already proficient in the target language. Furthermore, the relatively new nature of the technology may result in limited documentation and community support, leading to challenges in troubleshooting and resource availability. Debugging complexities arise from the intricate nature of the transpilation process, involvement of multiple technologies, and generation of multiple layers of code, leading to longer development and testing times. Lastly, the design's reliance on standard libraries in target languages may present limitations in functionality or necessitate additional development efforts to recreate or find alternative solutions, potentially adding complexity and increasing project costs.

Figure 19. Simplified sheet.

6.2. Future Work

The validation process of this proposal should involve several steps, including internal testing, case studies, benchmarking, and evaluation. Testing involves verifying correctness and completeness, whereas benchmarking involves comparing performance with other solutions. In addition, it is important to validate the proposal in industry, community, and expert forums. However, this is not a static process; the evolution of our proposal involves refining the design to make it more effective and adaptable to the defined scenarios.

Possible future work to further evaluate the effectiveness and validity of the proposed conceptual model for software design using a transpiler in the back-end layer could include the following:

- Conducting a new laboratory-based experiment with different functional requirements or approaches.
- Conducting a case study with a larger and more diverse sample size in order to gather feedback from a wider range of developers and end users.
- Identifying the potential limitations and weaknesses of the proposed model and proposing solutions to address them.
- Further benchmarking tests conducted using different software design models to validate the performance, scalability, and reliability of the proposed model.

- Developing a set of best practices and guidelines for effective use of the proposed model.
- Exploring the potential of the proposed model in different application domains and identifying scenarios in which it may be particularly effective.
- Investigating the feasibility of integrating the proposed model with other software design patterns and techniques.
- Developing tools and frameworks to facilitate the adoption and implementation of the proposed model.

Author Contributions: Conceptualization, A.B.F.; Methodology, A.B.F.; Software, A.B.F.; Validation, M.P.; Formal analysis, A.B.F., M.P. and J.M.; Investigation, M.P. and J.M.; Resources, M.P. and J.M.; Writing—original draft, A.B.F.; Writing—review & editing, A.B.F.; Visualization, M.P. and J.M.; Supervision, M.P. and J.M.; Project administration, M.P. and J.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by Smartwork S.A. company.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare that this study received funding from Smartwork S.A. company. The funder was not involved in the study design, collection, analysis, interpretation of data, the writing of this article or the decision to submit it for publication.

References

1. Kazman, R.; Klein, M.; Barbacci, M.; Longstaff, T.; Lipson, H.; Carriere, J. The architecture tradeoff analysis method. In Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193), Monterey, CA, USA, 14 August 1998 . [CrossRef]
2. Shaw, M.; Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, 1st ed.; Prentice Hall: Hoboken, NJ, USA, 1996.
3. Selić, B. Specifying dynamic software system architectures. *Softw. Syst. Model.* **2021**, *20*, 595–605. [CrossRef]
4. Fowler, M. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*; Addison-Wesley: Boston, MA, USA, 2012.
5. Erl, T. *SOA Principles of Service Design (the Prentice Hall Service-Oriented Computing Series from Thomas Erl)*; Prentice Hall PTR: Hoboken, NJ, USA, 2007.
6. Newman, S. *Building Microservices*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2021.
7. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*; John Wiley & Sons: Hoboken, NJ, USA, 2007.
8. Gahlot, S.; Chhillar, S.C. Design and Implementation of Component based Metric for Software Complexity Measurement. *Int. J. Recent Technol. Eng.* **2019**, *8*, 1093–1098. [CrossRef]
9. Yong, D. Design and Practice of Software Architecture in Agile Development. *J. Phys. Conf. Ser.* **2021**, *2074*, 012008. [CrossRef]
10. Blank, D.; Kay, J.S.; Marshall, J.B.; O'Hara, K.; Russo, M. Calico: A multi-programming-language, multi-context framework designed for computer science education. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, Raleigh, NC, USA, 29 February–3 March 2012. [CrossRef]
11. Mayer, P.; Kirsch, M.; Le, M.A. On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers. *J. Softw. Eng. Res. Dev.* **2017**, *5*, 1. [CrossRef]
12. Kargar, M.; Isazadeh, A.; Izadkhah, H. Improving the modularization quality of heterogeneous multi-programming software systems by unifying structural and semantic concepts. *J. Supercomput.* **2019**, *76*, 87–121. [CrossRef]
13. Kargar, M.; Isazadeh, A.; Izadkhah, H. Multi-programming language software systems modularization. *Comput. Electr. Eng.* **2019**, *80*, 106500. [CrossRef]
14. Izadkhah, H.; Kargar, M.; Isazadeh, A. Towards Comprehension of the Multi-Programming Language Software Systems. In Proceedings of the 2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI), Tehran, Iran, 28 February–1 March 2019. [CrossRef]
15. Kontogiannis, K.; Linos, P.; Wong, K. Comprehension and Maintenance of Large-Scale Multi-Language Software Applications. In Proceedings of the 2006 22nd IEEE International Conference on Software Maintenance, Philadelphia, PA, USA, 24–27 September 2006; pp. 497–500. [CrossRef]
16. Li, Z.; Qi, X.; Yu, Q.; Liang, P.; Mo, R.; Yang, C. Multi-Programming-Language Commits in OSS: An Empirical Study on Apache Projects. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021. [CrossRef]

17. Bastidas Fuertes, A.; Pérez, M.; Meza Hormaza, J. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. *Appl. Sci.* **2023**, *13*, 3667. [CrossRef]
18. Bastidas Fuertes, A.; Perez, M. A systematic review on Transpiler usage for Transaction-Oriented Applications. In Proceedings of the 2018 IEEE 3rd Ecuador Technical Chapters Meeting ETCM , Cuenca, Ecuador, 15–19 October 2018. [CrossRef]
19. Bierman, G.; Abadi, M.; Torgersen, M. Understanding TypeScript. In *ECOOP 2014—Object-Oriented Programming*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 257–281. [CrossRef]
20. Grant, M.J.; Booth, A. A typology of reviews: An analysis of 14 review types and associated methodologies. *Health Inf. Libr. J.* **2009**, *26*, 91–108. . [CrossRef]
21. Grichi, M.; Abidi, M.; Jaafar, F.; Eghan, E.E.; Adams, B. On the Impact of Interlanguage Dependencies in Multilanguage Systems Empirical Case Study on Java Native Interface Applications (JNI). *IEEE Trans. Reliab.* **2021**, *70*, 428–440. [CrossRef]
22. Vraný, J.; Píše, M. Multilanguage debugger architecture. In *SOFSEM 2010: Theory and Practice of Computer Science*; Springer: Berlin/Heidelberg, Germany, 2010; Volume 5901, pp. 731–742. [CrossRef]
23. Neitsch, A.; Wong, K.; Godfrey, M.W. Build system issues in multilanguage software. In Proceedings of the IEEE International Conference on Software Maintenance, ICSM, Trento, Italy, 23–28 September 2012; pp. 140–149. [CrossRef]
24. Vinoski, S. Multilanguage programming. *IEEE Internet Comput.* **2008**, *12*, 83–85. [CrossRef]
25. Mayer, P.; Bauer, A. An empirical analysis of the utilization of multiple programming languages in open source projects. In Proceedings of the ACM International Conference Proceeding Series, Cape Town, South Africa, 27–29 April 2015. [CrossRef]
26. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Patterns, D. Elements of Reusable Object-Oriented Software. *Des. Patterns* **1995**, 293–303 .
27. Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; Stafford, J. *Documenting Software Architectures: Views and beyond*; Addison-Wesley Professional: Boston, MA, USA, 2010.
28. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*; Addison-Wesley Professional: Boston, MA, USA, 2015.
29. Brown, M.; Wilson, A. *Fundamentals of Software Architecture: An Engineering Approach*; O'Reilly Media: Sebastopol, CA, USA, 2018.
30. Garlan, D.; Shaw, M. *Software Architecture*; CRC Press: Boca Raton, FL, USA, 2016.
31. Kruchten, P. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*; Addison-Wesley Professional: Boston, MA, USA, 2019.
32. Adel, A.; Adel, S. Integration of Architectural Design and Implementation Decisions into the MDA Framework. In Proceedings of the Third International Conference on Software and Data Technologies 2008, Porto, Portugal, 5–8 July 2008; Volume 1: ICSOFT, pp. 366–371. [CrossRef]
33. Alti, A.; Boukerram, A.; Roose, P. Context-Aware Quality Model Driven Approach: A New Approach for Quality Control in Pervasive Computing Environments. In Proceedings of the Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, 23–26 August 2010; Babar, M.A., Gorton, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 441–448.
34. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley Professional: Boston, MA, USA, 2002.
35. Al-Hawari, F. Software design patterns for data management features in web-based information systems. *J. King Saud Univ. -Comput. Inf. Sci.* **2022**, *34*, 10028–10043. [CrossRef]
36. Cervantes, H.; Kazman, R. *Designing Software Architectures: A Practical Approach*; Addison-Wesley Professional: Boston, MA, USA, 2016.
37. Stahl, T.; Völter, M.; Czarnecki, K. *Model-Driven Software Development: Technology, Engineering, Management*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2006.
38. Velepucha, V.; Flores, P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access* **2023**, *11*, 88339–88358. [CrossRef]
39. Bisson, R. SQL injection. *ITNOW* **2005**, *47*, 25–25. [CrossRef]
40. Cabibbo, L.; Carosi, A. Managing Inheritance Hierarchies in Object/Relational Mapping Tools. In *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*; Springer International Publishing: Berlin/Heidelberg, Germany, 2005; pp. 135–150. [CrossRef]
41. Microsoft. Entity Framework. 2022. Available online: https://learn.microsoft.com/en-us/aspnet/entity-framework (accessed on 2 October 2023).
42. Bastidas Fuertes, A.; Pérez, M. Transpiler-based architecture for multi-platform web applications. In Proceedings of the 2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM), Salinas, Ecuador, 16–20 October 2017; pp. 1–6. [CrossRef]
43. Cusumano, M. Company character and the software business. *Commun. ACM* **2003**, *46*, 21–23. [CrossRef]
44. Hemphill, T.A. Government Technology Acquisition Policy: The Case of Proprietary Versus Open Source Software. *Bull. Sci. Technol. Soc.* **2005**, *25*, 484–490. [CrossRef]
45. Ernst, N.A.; Klein, J.; Bartolini, M.; Coles, J.; Rees, N. Architecting complex, long-lived scientific software. *J. Syst. Softw.* **2023**, *204*, 111732. [CrossRef]
46. Bennett, K.; Munro, M.; Gold, N.; Layzell, P.; Budgen, D.; Brereton, P. An Architectural model for service-based software with ultra rapid evolution. In Proceedings of the IEEE International Conference on Software Maintenance, ICSM 2001, Florence, Italy, 7–9 November 2001. [CrossRef]
47. Mathieu, R.G.; May, J.L.; Reif, H.L. Investigating open source software creators through the lens of an entrepreneur. *Int. J. Innov. Learn.* **2017**, *21*, 1. [CrossRef]

48. Galster, M.; Weyns, D. Empirical research in software architecture—Perceptions of the community. *J. Syst. Softw.* **2023**, *202*, 111684. [CrossRef]
49. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2012. [CrossRef]
50. Staron, M. *Action Research in Software Engineering*; Springer International Publishing: Berlin/Heidelberg, Germany, 2020. [CrossRef]
51. Schultes, D. SequalsK—A Bidirectional Swift-Kotlin-Transpiler. In Proceedings of the 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems, MobileSoft 2021, Madrid, Spain, 17–19 May 2021; pp. 73–83. [CrossRef]
52. Cordasco, G.; D'Auria, M.; Negro, A.; Scarano, V.; Spagnuolo, C. FLY: A Domain-Specific Language for Scientific Computing on FaaS. In *Euro-Par 2019: Parallel Processing Workshops*; Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Springer: Cham, Switzerland, 2020; Volume 11997, pp. 531–544. [CrossRef]
53. Mudalige, G.; Giles, M.; Reguly, I.; Bertolli, C.; Kelly, P. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In Proceedings of the 2012 Innovative Parallel Computing (InPar), San Jose, CA, USA, 13–14 May 2012; pp. 1–12. [CrossRef]
54. Moses, W.S.; Ivanov, I.R.; Domke, J.; Endo, T.; Doerfert, J.; Zinenko, O. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, Montreal, QC, Canada, 25 February–1 March 2023; pp. 119–134. [CrossRef]
55. Yue, S. Program Transformation Techniques Applied to Languages Used in High Performance Computing. In Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, Indianapolis, IN, USA, 26–31 October 2013; pp. 49–52. [CrossRef]
56. Teixeira, G.; Bispo, J.A.; Correia, F.F. Multi-Language Static Code Analysis on the LARA Framework. In Proceedings of the 10th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, Virtual, 22 June 2021; pp. 31–36. [CrossRef]
57. Klingler, R.; Trifunovic, N.; Spillner, J. Beyond @CloudFunction: Powerful Code Annotations to Capture Serverless Runtime Patterns. In Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021, Virtual, 6–10 December 2021; pp. 23–28. [CrossRef]
58. Li, Z.; Wang, S.; Wang, W.; Liang, P.; Mo, R.; Li, B. Understanding Bugs in Multi-Language Deep Learning Frameworks. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, Australia, 15–16 May 2023; pp. 328–338. [CrossRef]