Modular and Distributed IDE

Fabien Coulon
fabien.coulon@obeo.fr
Obeo
France
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Alex Auvolat
alex.auvolat@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Benoit Combemale
benoit.combemale@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Yérom-David Bromberg
david.bromberg@irisa.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

François Taïani
francois.taiani@irisa.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Olivier Barais
olivier.barais@irisa.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Noël Plouzeau
noel.plouzeau@irisa.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Abstract

Integrated Development Environments (IDEs) are indispensable companions to programming languages. They are increasingly turning towards Web-based infrastructure. The rise of a protocol such as the Language Server Protocol (LSP) that standardizes the separation between a language-agnostic IDE, and a language server that provides all language services (*e.g.,* auto completion, compiler...) has allowed the emergence of high quality generic Web components to build the IDE part that runs in the browser. However, all language services require different computing capacities and response times to guarantee a user-friendly experience within the IDE. The monolithic distribution of all language services prevents to leverage on the available execution platforms (e.g., local platform, application server, cloud). In contrast with the current approaches that provide IDEs in the form of a monolithic client-server architecture, we explore in this paper the modularization of all language services to support their individual deployment and dynamic adaptation within an IDE. We evaluate the performance impact of the distribution of the language services across the available execution platforms on four EMF-based languages, and demonstrate the benefit of a custom distribution.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: Microservice, IDE, Generative approach

ACM Reference Format:
Fabien Coulon, Alex Auvolat, Benoit Combemale, Yérom-David Bromberg, François Taïani, Olivier Barais, and Noël Plouzeau. 2020. Modular and Distributed IDE. In *Proceedings of Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SLE '20, November 16–17, 2020, Virtual, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern Integrated Development Environments (IDE) are moving to the Software as a Service (SaaS) model [2] in order to benefit from advantages [7] such as better accessibility (since the application is already installed and configured), lower costs to maintain and/or upgrade, scalability, etc. The rise of a protocol such as the Language Server Protocol (LSP) that standardizes the protocol used between a language-agnostic IDE and a language server that provides *language services* (as defined in Definition 1.1) such as auto completion, search for definition, search for all references, compilation, etc. has allowed the emergence of high quality generic Web components to build the IDE part that runs in the browser. For instance, Monaco[1] (used in VSCode[2], Theia[3] , …), Atom[4], CodeMirror[5] (CodePen[6], Jupyter[7]), are now embeddable Web components with a direct support of most of the LSP features, thereby simplifying the development of Web-based IDEs.

Definition 1.1. We consider as *language service* any language-specific functionality that can be used by a language user, and that takes a program as input. Such service can be a functionality related to editing programs giving quick feedback to the user (e.g., auto completion, goto definition, etc) but can also be a more long running functionality related to other activities (e.g., compiler, debugger, etc).

[1] https://microsoft.github.io/monaco-editor/
[2] https://code.visualstudio.com/
[3] https://theia-ide.org/
[4] https://atom.io/
[5] https://codemirror.net/
[6] https://codepen.io/
[7] https://jupyter.org/

However, defining the architecture of an LSP server implementation and more generally the server implementation for a particular language remains a complex step. The simplistic deployment of the language server part in a sufficiently powerful cloud does not in reality provide the optimal user-experience. Each of language services has specific requirements in terms of latency and bandwidth, but also in terms of specific computing capacity. It is therefore important to tune the deployment according to the services of a particular language but also according to the context of use of the IDE for a given user, and the available execution platforms. For example, it could be required to reduce the network requirements if the quality of the network decreases for a specific user. Such an implementation of a language server should therefore be essentially a Dynamically Adaptive System (DAS)[9] in which we could provide tailored distribution of the language services that optimizes the user experience and their overall performance. Defining the architecture of such a system requires fine-grained modularity in both design and deployment, and the ability to run in a distributed and heterogeneous environment.

In contrast with the current approaches that provide IDEs in the form of a monolithic client-server architecture, we explore in this paper the modularization of all language services to support their individual deployment and dynamic adaptation within an IDE. Since the distribution requires both the modularization of the language services, and the deployment (and possibly the dynamic reconfiguration) over the available environment, we propose a generative approach to automatically obtain microservices implementing language services from a language specification, complemented with a feature model that drives the safe configuration and automates the deployment of IDE features. We explicit *IDE feature* in Definition 1.2.

Definition 1.2. We define an *IDE feature* as the deployment unit of a distributed IDE. It is a coherent group of *language services* intended to always be deployed together.

We study the impact on performances when distributing the language services across the available execution platforms. We evaluate our approach on four EMF-based languages and demonstrate the benefit of a custom distribution of the various language services. In particular, we apply our approach to NabLab[8], our own implementation of the Logo language[9], MiniJava[10] and ThingML[11] to compare response times of language services in our approach with monolithic language server.

This paper is organized as follows. Section 2 motivates this work by describing the example of a monolithic language server, and the heterogeneity of the different proposed language services. Section 3 gives the big picture of our approach and explains why we use microservices and feature modeling. Section 4 describes how we generate microservices, as well as the feature model and its associated deployment approach. In section 5 we study our approach on four EMF-based languages and we evaluate the cost by comparing the response times of the language services of our microservice-based architecture with that of monolithic language servers.

2 Motivating example

To illustrate the heterogeneity of the various services provided by modern IDEs, we use in this section and throughout this paper the open-source and industrial Domain-Specific Language (DSL) NabLab. NabLab provides a productive development environment for numerical analysis over exascale HPC technologies. The associated IDE provides all the common editing services (syntax coloring, auto-completion, validators...) and a complex compilation chain targeting various backends. NabLab users are mathematicians and physicists that write algorithms for numerical analysis. NabLab programs are mainly composed of jobs with complex data flow between them, representing physical systems. As the computation used to simulate physical systems is expensive, NabLab programs are given to a compilation chain generating efficient source code to run the different jobs in parallel.

Figure 1 represents the current IDE architecture according to the state of practices. This architecture separates the IDE client, which is the language-agnostic interface for language users, from the language server which implements the language-specific services. The IDE client remotely calls these services by sending JSON-RPC[12] messages to the language server. This architecture allows to deploy the IDE client and the language server possibly in different execution platforms (*e.g.,* the client on the development laptop, and the server on the cloud or an application server). In practice, NabLab has been developed using the *Eclipse Modeling Framework* [10], including the Ecore[13], Xtext[14] and Sirius[15] technologies.

For the sake of illustration, we selected four representative language services provided by the NabLab IDE: *completion* is a content assist that returns a list of proposals for a given context, *references* searches for elements in a file referring to a given symbol, *rename* changes names for a given element and for all its referring elements, and *compiler* performs graph analysis of the concurrent job to generate optimized

[8] https://github.com/cea-hpc/NabLab
[9] https://github.com/fcoulon/sle2020-dev/tree/master/logo/logo.xtext.parent
[10] https://github.com/tetrabox/minijava
[11] https://github.com/TelluIoT/ThingML
[12] https://www.jsonrpc.org/
[13] https://www.eclipse.org/ecoretools
[14] https://www.eclipse.org/Xtext
[15] https://www.eclipse.org/sirius

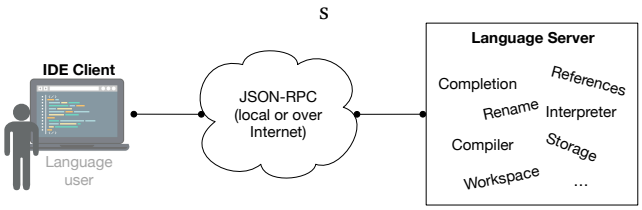

Figure 1. Current IDE Architecture, including a language-agnostic client that provides the user interface, and a language-specific server that provides all the language services for a given language

Java source code (one of the possible backends in NabLab). We have chosen these language services to be representative of language user's activities: editing of code, navigation in the code, code refactoring, and code transformation.

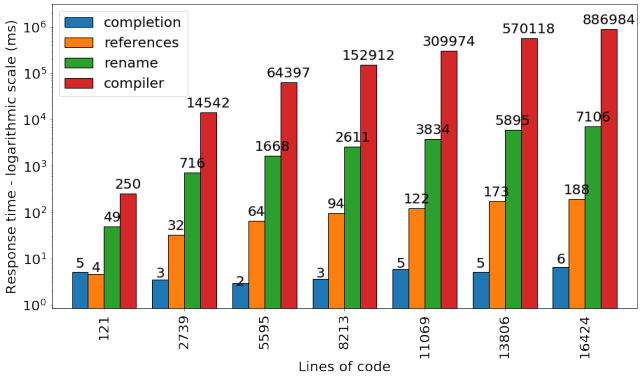

Figure 2. Response times of four NabLab services (client and server deployed on the same local development laptop)

In the NabLab IDE, *completion*, *references* and *rename* are obtained with Xtext and the support of the *Language Server Protocol* (LSP), while *compiler* is a separate compilation chain integrated and prompted from the IDE. To illustrate the heterogeneity of these language services and the potential benefits of distributing them, we measured their response times on NabLab files of increasing size. The measurements were performed 100 times for each language service, with a client and a server both deployed on the same machine with an Intel Core i7-7600U CPU at 2.80GHz, 32 GiB of RAM, and the HotSpot JVM 11.0.5. Figure 2 presents the means of the response times expressed in milliseconds (ms) for the different files and, due to the large range of values, with a logarithmic scale.

We observe significant heterogeneity among language services, ranging from *completion* that lasts about 5 ms constantly over the files, to *references* that goes up to about 188 ms, *rename* that goes up to 7.106 seconds, and finally *compiler* that goes up to 14.78 minutes. This difference in response times between the language services is of several orders of magnitude. It can be explained by the amount of computation performed by each service: *completion* traverses object's references, *reference* is a query in a graph of objects, *rename* rewrites the file and *compiler* performs complex graph analysis and generates source code. This motivates the need for an individual and distributed deployment of each language service to leverage better the available execution platforms, fit the activities performed, and eventually provide the best user experience within the IDE client. In particular, in this paper, we focus on the following research questions:

RQ1 Is it possible to provide a systematic approach that automates the modularization of the language service implementations, supports their individual deployment, and enables their dynamic adaptation according to a given context (*e.g.* usage, environment)?

RQ2 Is it possible to optimize the distribution of the language services across the available runtime platforms (*e.g.,* local platform, application server, cloud) to improve their performances within the IDE?

3 Approach overview

We propose a systematic approach that eases the modularization and distribution of highly configurable IDEs for DSLs. The approach takes as inputs i) a software language specification, in the form of a metamodel, a syntax description, and any additional concerns such as validators, compilers, etc., and ii) a set of desired features (*i.e.,* coherent groups of services) that the IDE must or may provide. As output, the approach generates a set of modular, language-specific, IDE features and a tool-supported feature model to configure and automate their distribution and integration within a Web-based IDE.

We distinguish two different user roles: language users, and language designers. In our process, configurability of an IDE by an end user relies on software product line principles: using a language specification as input, language designers build in fact a family of distributed IDEs. Language designers, or even language users, can then configure the family to deploy a distributed IDE that suits the user's needs and experience. Since the needs of language user can evolve over time, the deployment of the IDE can be dynamically reconfigured

Figure 3 presents the overall approach, from the specifications to the deployment of an IDE. First, a language designer provides a *language specification* along with a *protocol specification* that describes the expected modularity of the language services and their interactions. From these specifications, we automatically generate a set of microservices implementing the IDE features and a feature model that captures their valid configurations (step ①). The feature model offers a description of the variability of a system, here our variability is the availability of IDE features and their deployment configuration, presented as a tree of features enriched with logical constraints. It essentially describes what the feature

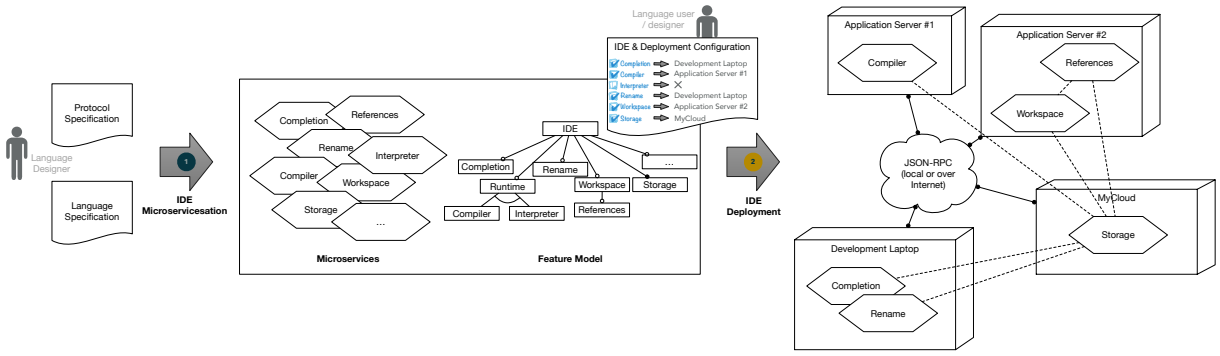

Figure 3. Approach overview, with the two main steps: ① IDE microservicesation and ② IDE deployment

alternatives are, their dependencies and whether they are mandatory or optional. Since dependencies form a graph, we compute a minimum spanning tree to create the hierarchy of the feature model and we encode as logical constraints the remaining dependencies. The IDE and its deployment can then be configured by a language designer or user (step ②), depending on who has the knowledge to decide where to deploy the microservices. Configurations could also be proposed by an automated process involving a predictive model in the deployment, or through dynamic reconfiguration, in an attempt to maintain metrics such as user experience. The IDE configuration consists of selecting the microservices that will be available to the language user, together with information on where to deploy them. The feature model is used to validate the set of microservices to be deployed by checking the variability constraints. At the end, the microservices are distributed to different execution platforms such as development laptop, cloud or application servers. At this stage, the language services are running and the language user can use them through an IDE client.

The following quality criteria guided our current implementation of this systematic approach:

- IDE configurability for the end user: it is supported by a feature model driving the safe deployment of the language services
- efficiency of resource usage (CPU time, bandwidth, reactivity as perceived by the end user): it is allowed by the modularization of language services which are individually deployables according to their expected optimal quality of service
- extendibility and reusability from the point of the language designer, *i.e.,* when the set of features evolve or when distributed platform technologies change (long term, human driven adaptation): they are enabled thanks to the specification of the protocol by the language designer which defines the granularity of the language services and their dependencies
- adaptability, possibly dynamically (according to the usage and environment): it is reached through the stateless nature of generated microservices that supports dynamic reconfiguration (maintenance, evolution, deployment, ...)

Our process design decisions were taken to obtain a satisfactory balance of these criteria.

In the rest of this section, we detail the two main steps of our approach (① and ② in Figure 3).

3.1 Designing IDE microservices

The design of an IDE family is based on two main inputs that are required to improve flexibility and reusability of elementary design elements, thereby supporting the language designer in providing a highly and dynamically customizable distributed IDE.

Language specification. The language designer needs a language specification. In our prototype, the language specification comes as an Ecore metamodel, and Xtext grammar description, and additional services such as compilers. Using tools such as Xtext, the language designer is able to produce a software module that acts as a parser and builds a model from a program file, as an internal, metamodel compliant, form of the program.

Protocol specification. The language designer also provides a description of the expected modularity of the language services in the form of a so called *protocol*. Code completion, symbol renaming, compiler, are examples of such language services (*capabilities*) that may be grouped into IDE features as deployment units. The grouping of the language services into IDE features, and their inter-dependencies, are expressed in a specific model, for which we provide a specific DSL with a concrete syntax, and a metamodel shown in Figure 4.

Using this DSL the language designer is able to declare the properties and relationships of each feature declared in a protocol specification. The DSL relies on the following types (cf. Figure 4):

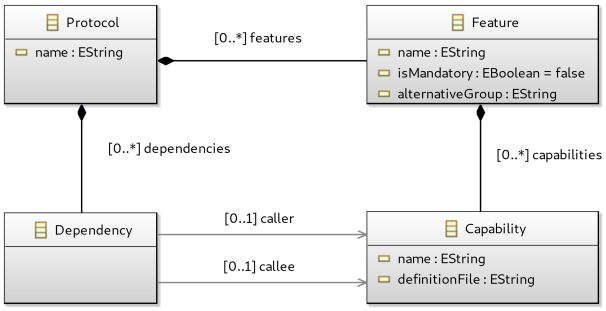

Figure 4. The metamodel used to describe a protocol for IDE features

- The Capability type allows for the definition of a basic service point, akin to a callable function in the architecture. Details of the function implementation, such as a port number for micro-service based implementations are also indirectly provided by a Capability.
- The Feature type regroups a coherent set of capabilities. It typically represents an elementary tool supported by the IDE (*e.g.,* code completion, name refactoring).
- The Protocol type regroups the features that are potentially supported by an IDE.
- The Dependency type describes the relationships between capabilities supported by a protocol, for instance the call dependency between two capabilities.

From the language specification and the protocol specification, we implemented a generative approach (① in Figure 3) that produces:

- a set of modular language services in the form of cloud-native applications as microservices, and the required code that will take care of the communication between language services as described in the protocol.
- a feature model that represents the IDE family.

The feature model can then be enriched with deployments constraints, *e.g.*:

- for efficiency reason some capabilities need to be implemented by the same feature, and therefore deployed on the same execution node;
- some features are alternatives in a group, *e.g.,* if at run time the currently deployed feature becomes unavailable then one alternative will be deployed automatically.

3.2 IDE Deployment

The final task to build a usable, running IDE is the deployment phase. As mentioned before, we aim at providing a family of IDEs to the final language user, as different users have different needs, and a given user may also wish to tailor the IDE depending on the current tasks she is involved in. To support this flexibility we provide a deployment configurator, which is parameterized by the feature model generated from the protocol specification, to allow user control on which capabilities to deploy and where, while maintaining the constraints defined in the feature model.

A point to consider in order to support reconfiguration of the deployment (*i.e.,* to move microservices) is that we implement language services as stateless microservices. This has benefits for the scalability of distributed applications since a microservice can be replicated and the requests dispatched among the different instances to handle load increases. It also allows the microservices to be moved easily from one location to another, and it avoids data loss in case of a microservice crash. However, stateless microservice does not keep any states between requests that involve retrieving programs from a persistence storage and parsing them to get a model before processing language services, which increases response times compared to stateful microservices. This additional cost on response times must be taken into account to benefit from the reconfiguration of a distributed IDE.

The proposed generative approach (② in Figure 3) takes as inputs a specific configuration of the feature model and the microservices, and produces a distributed IDE integrated with the Web-based client.

4 Towards a modular and distributed NabLab IDE

In this section we use a running example to detail the steps introduced in the previous section. Our task is the design of a distributed IDE for Nablab users. We chose NabLab as a language for our experiments on modular and distributed IDE construction because developing NabLab software provides a wide range of requirements, from responsive editing operations of code to CPU intensive compilation and execution. Taking care of this range of requirements is best addressed by distributing language services on various types of execution platforms.

NabLab users are supported by a set of tools that form a specific IDE:

- A textual editor supports contextual code completion, code folding, syntax highlighting, error detection, quick fixes, variable scoping, and type checking.
- A model explorer provides a dedicated outline view and a contextual LaTeX view.
- A debugging environment provides variable inspection, plot display and 2D/3D visualization.
- A NabLab compiler generates efficient implementations thanks to the associated compilation chain.

4.1 Language and protocol specifications

As mentioned in the previous section, in our approach a language specification consists of defining a metamodel, a concrete syntax and semantics. We use the Eclipse Modeling

```
Protocol {
  mandatory feature storage {
    capabilities :
      document
      update
  }
  feature completion {
    capabilities:
      complete
  }
  (...)
  dependencies {
    completion.complete -> storage.document
    (...)
  }
}
```

Listing 1. Excerpt of the protocol specification for the completion feature of NabLab

Framework[16] (EMF) and its ecosystem to define our language specification.

The concrete syntax of NabLab is a grammar defined with Xtext[5]. Taking a grammar as input, Xtext is able to generate the source code implementing a set of language services for a text editor. Xtext can also generate a language server, which embeds the language services that are then callable remotely. In our approach we use this generator to produce implementations of IDE features.

The compilation chain comes as a separate language service implemented with Xtend[17], and this service is integrated and prompted from the IDE client.

Listing 1 is an excerpt from the protocol specification for NabLab. This protocol specification conforms to the metamodel described in Figure 4. Our tool to edit a protocol textual specification and simultaneously build a protocol model is based on Xtext. As a language designer, through the protocol specification we made the choice for NabLab to define the language services as stateless services and we defined the granularity of the deployment units by declaring how the language services are grouped in the IDE features. Our NabLab protocol model declares the *storage* and *completion* features, and a dependency between their respective capabilities. The *storage* feature provides the following capabilities: *document*, to retrieve a persisted document (a program), and *update*, to change the contents of a program. The *completion* feature provides the *complete* capability to get content assist proposals. In the dependencies, the *complete* capability first retrieves a program by calling the *document* capability before computing the set of completion choices. The protocol specification of NabLab, in addition to the *storage* and *completion* features shown in the listing, also has the following features: *workspace*, which computes diagnostics for programs and indexes their content, *definition*, which gives the location of an element's definition, *highlight*, which looks up the element's definition as well as other elements linked to the same definition, *hover*, which returns the element's description, *documentSymbol*, which returns all the elements of a program, *formatting*, which computes text edition operations to normalize program's indentation, *rename*, which returns text edition operations to rename an element's definition and other elements linked to the same definition, *references*, which finds elements having the same definition, *symbol*, which returns all the symbols in the opened program matching a query, and *compiler*, which generates a Java source file from a NabLab file.

4.2 Feature model generation

In a second step we generate a NabLab feature model by taking the NabLab protocol specification as input. Figure 5 presents the resulting feature model built by applying the approach described in the previous section. The features of the model match the IDE features of the protocol specification and the hierarchy is derived from the dependencies between capabilities. For example, in the protocol specification *symbol* has a capability calling the *index* capability declared in *workspace*. We infer that *symbol* requires *workspace* to run, and we set a parent-child relationship in the feature model. It means that a configuration of this feature model containing *symbol* is valid only if *workspace* is also present. We compute similarly the hierarchy for other features.

Our implementation of the feature model generator relies on the Feature IDE framework [13], which provides facilities to construct and manipulate feature models.

The feature model generated with this framework is then used to configure the expected deployment and ensure its validity with regards to the constraints defined in the feature model.

4.3 Microservice generation

We use the Quarkus[18] framework to implement Java microservices for each IDE feature defined in the protocol specification. A Java class is generated with methods corresponding to the *Capabilities* of the IDE feature. Each *Capability* is implemented as a REST API, by annotating methods with HTTP verbs, query parameters, and endpoint paths. These methods are callbacks for the HTTP requests. We process the Capability's JSON definition file to generate arguments and return type for the methods. We also generate proxy classes from the dependencies of the IDE features. These classes provide methods to remotely call the capabilities implemented in the other microservices.

The language service implementations of NabLab are provided by Xtext, except for the compiler. We leverage on

[16] https://www.eclipse.org/modeling/emf/
[17] https://www.eclipse.org/xtend/
[18] https://quarkus.io/

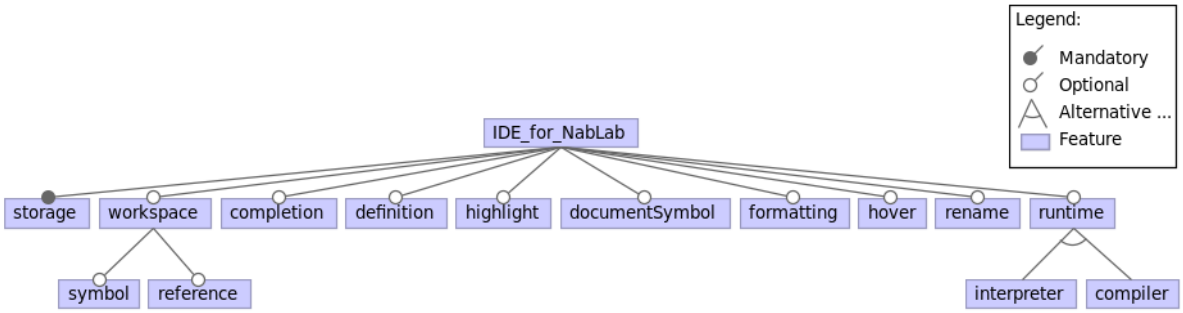

Figure 5. Feature model of the language NabLab

these available implementations by providing a generative approach that reuses the Xtext generators to complete microservice implementations. The integration with an IDE client is not addressed by this work and it is up to the language designer to implement the calls to the microservices from the client. However, this can be facilitated by using an integration protocol such as the LSP that provides interoperability between client IDEs and language services.

4.4 Deployment configuration

Our distribution of the IDE features relies on Docker[19] and the container orchestrator Kubernetes[20]. The microservices are packaged in Docker containers, which isolate the microservices and ease the deployment by embedding their runtime environment (*e.g.,* a JVM). We deploy containers in a cluster by using a container orchestrator, since it allows to plan their deployment at different locations and move them at runtime. Our deployment configurator monitors the deployment using the Kubernetes API, to get the list of deployed microservices with their locations to construct the configuration representing the current deployment. We provide a frontend for the configurator as a web page that displays configurations to the language user and designer. Through this web page, the language user can change the deployment configuration by disabling microservices, selecting new ones and change their deployment location. If the new configuration is valid, a deployment plan is sent to the Kubernetes API. Although we implemented our deployment configurator with Kubernetes, other container orchestrators like Nomad[21] can be used to distribute IDE features (as shown in section 5 where we manually deploy IDE features on a Nomad cluster)

5 Experimentations

In this section we present our experiments to answer the research questions introduced in Section 2 (RQ1 and RQ2). We conducted these experiments on four EMF-based languages: NabLab, our own implementation of the Logo language, ThingML and MiniJava. Logo is an educational programming language dedicated to 2D drawings, ThingML is dedicated to applications for the Internet of Things, and MiniJava is a subset of the Java language. We selected this mix of general purpose and domain specific languages of distinct domains to be representative of the generalization of the proposed approach.

Our experiments compare language services implemented by monolithic servers deployed locally and modular servers distributed over the available execution platforms. Our results show that monolithic servers deployed locally (*i.e.,* on the same execution platform than the client) offer better response times when not resource demanding but quickly limited for computationally intensive IDE features, and that parsing models and loading model elements constitute a bottleneck for microservices, which would prevent the design of an entirely stateless architecture.

5.1 Experimental setup

The evaluation was performed on three machines in a Nomad cluster. All machines were Dell PowerEdge R330 with Intel(R) Xeon(R) CPU E3-1280 v6 @ 3.90GHz 4 cores, hyper-threading and 31GB of RAM. The IDE features were deployed in Docker containers and running on OpenJDK 11.0.5.

The evaluation measured the response times of language services implemented as monolithic language servers, and as distributed language servers with microservices. Both are implementing the Language Server Protocol[22] (LSP), and possibly additional services such as a compiler. Monolithic language servers were deployed on a single machine of the cluster. We generated programs of a similar number of lines of code based on the content of existing examples for each language[23]. For each program, we initialized an LSP session and opened the program. After the initialization is completed, we then called 100 times each language service in a row and

[19] https://www.docker.com/
[20] https://kubernetes.io/
[21] https://www.nomadproject.io/
[22] https://microsoft.github.io/language-server-protocol/
[23] https://anonymous.4open.science/r/e03961ac-9f27-4c52-ab28-87cf105a83f4/

compute the average of the time elapsed between the request and the reply measured from the client's point of view.

We performed the same experiments with the distributed servers after distribution of the language services over the available cluster. On the first machine we deployed the language service *storage*, on the second machine *workspace*, *completion*, *definition*, *highlight*, *documentSymbol* and on the third machine *hover*, *references*, *rename*, *symbol*. Since we wanted to measure the cost of message exchanges, we deployed depending IDE features on different machines: for instance, *workspace* feature was not on the same machine as *references* and *symbol*, that *storage* is on its own machine. The client that sends the requests to language services was deployed outside of the cluster, on a machine connected to the cluster by a local network.

In the case of the distributed servers, we also measured the times taken to load and resolve the references of models, in order to highlight the impact of load and resolve phases in the response time of the microservices. More precisely, models are graphs of objects that reference each other but these references are not resolved at the first load, they are resolved on demand when language services browse the models, which has an impact on the processing time of the language services. For each language we repeated 100 times the measure of loading and resolution time of each program.

5.2 Results

To investigate the gains and the costs of distributing language services, we first compared the response times of language services of the motivating example in Section 2, which are implemented by a monolithic server running on a laptop (client co-located on the laptop), to the response times of the same language services implemented by microservices running on the Nomad cluster (client outside of the cluster, and connected by a local network) for the NabLab language. In a second experiment, we compared the response times of language services for monolithic and distributed architectures running both on the Nomad cluster to evaluate the cost of the distribution for the languages NabLab, Logo, ThingML and MiniJava. We then measured the loading time of the programs for these four languages and compare them to the response times of language services to evaluate the impact of the stateless architecture.

5.2.1 Microservice-based version. We measured the response times of microservices running on a cluster implementing the language services of the motivating example presented in Figure 2 and compare them with the response times of language services running locally on the laptop. Figure 6 presents response times of these microservices deployed in the Nomad cluster. The percentage given for each bar represents the overhead with regard to the time presented in Figure 2. Response times of the feature *completion* increase with file size and are several orders higher than

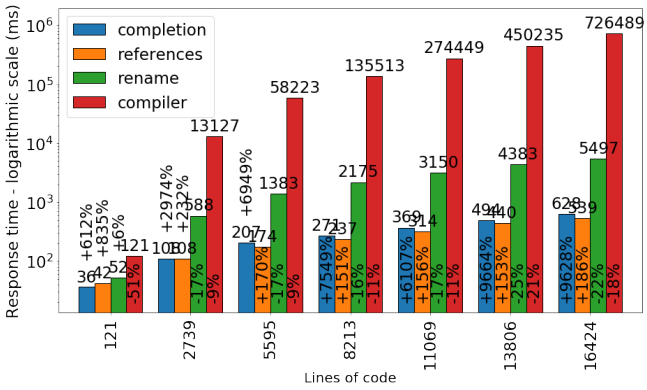

Figure 6. Response times of language services of the motivating example deployed as microservices in the Nomad cluster, and comparison with the response times from Figure 2.

in Figure 2, where they are between 2 and 6 milliseconds. Response times for *references* are 2 to 10 times higher than those in Figure 2. Feature *rename* takes 6% more on the 121-line file, but gains between 16% and 25% for the other files. Feature *compiler* gains 51% on the 121-line file and gains between 9% and 21% on the other files.

5.2.2 Protocol. To evaluate the cost of modularization and distribution of the language services, we measured response times on programs of increasing sizes for the languages Logo, NabLab, ThingML and MiniJava, with both monolithic and distributed servers. For the distributed architecture, we also measured the overheads from message exchanges between the microservices to fulfill the request to language services (*i.e.,* the total time spent by a microservice waiting for responses to requests for additional inputs from other microservices that are necessary to process its internal logic). We performed 100 measurements for each language service on each program and for both servers, and computed the means. For the 320 measured means, 232 of them have a coefficient of variation (*i.e.,* the ratio of the standard deviation to the mean) below 30%.

Figures 7, 8, 9 and 10 presents a comparison of the response times of the language services for the monolithic and distributed architectures on programs of increasing size, respectively for NabLab, Logo, MiniJava and ThingML. The cumulative times resulting from the modularization and the distribution of language services are represented by the *protocol overhead* parts, in grey in the response times of the distributed architecture. They measure the times taken by the exchanges between the distributed microservices, which include the retrieval of programs from the *storage* microservice, and the retrieval of the workspace's index (only for *references* and *symbol* features). The upper parts of the bar displayed in orange represent the times not due to the exchanges between the microservices, which include the processing times

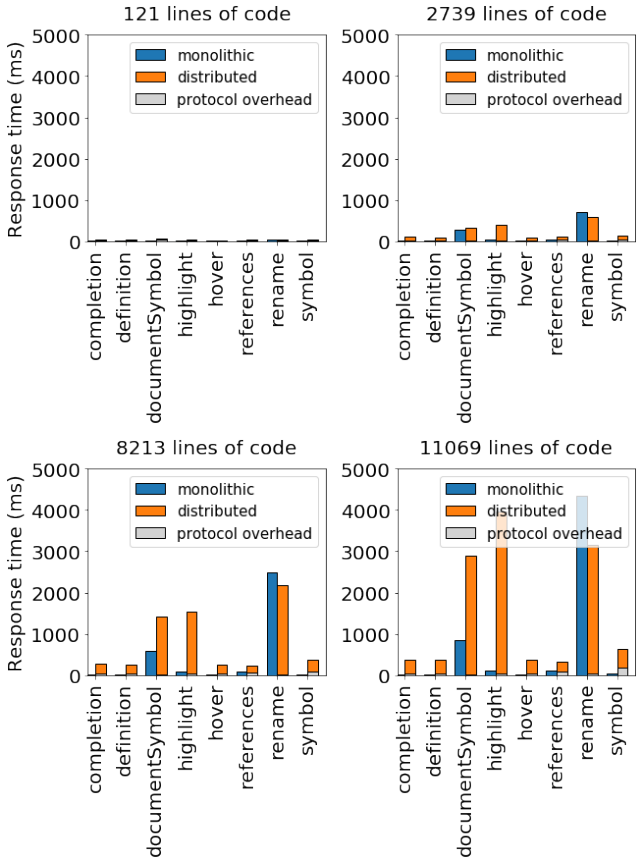

Figure 7. Comparison of response times in monolithic and distributed architectures for language services of NabLab

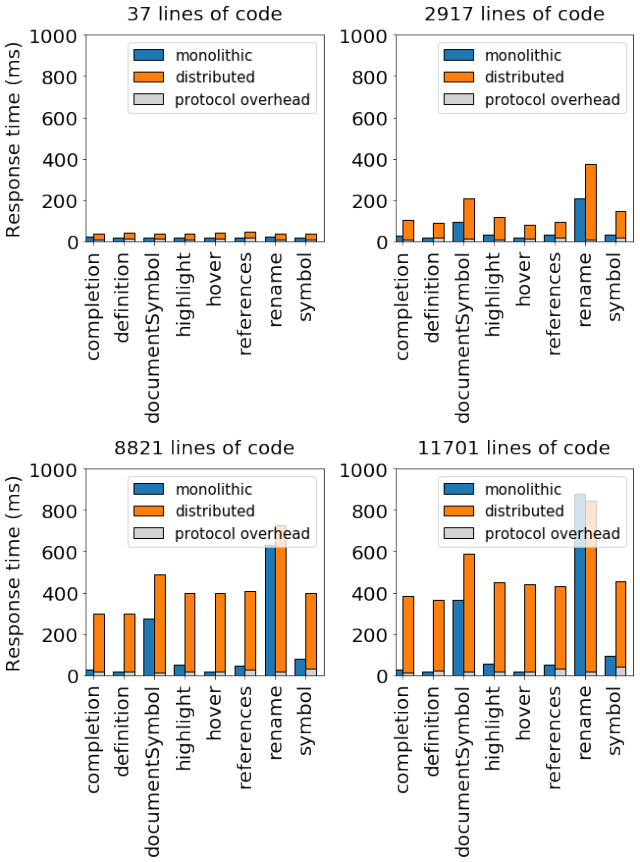

Figure 8. Comparison of response times in monolithic and distributed architectures for language services of Logo

of the internal logic of the microservice implementing the language service, and the time for sending the response to the client.

In most cases, the distributed architecture introduces an overhead in comparison to the monolithic implementation. However, we observe that the overhead is marginally due to the protocol (the means for each file are between 24.63 ms and 57.64 ms for NabLab, 11.43 ms and 22.88 ms for Logo, 14.21 ms and 21.94 ms for MiniJava, 9.31 ms and 15.96 ms for ThingML), but mostly lies in the orange parts and is rather due to the stateless nature of microservices. We further explore this in the rest of this section.

5.2.3 Statelessness impact. The microservices being stateless, they all require to fetch and load the necessary part of the model in addition to executing the corresponding language service(s). To evaluate the impact on response times of the stateless nature of microservices implementing language services, we measured the initial load times and the full references resolution times for the four languages NabLab, Logo, Minijava and ThingML on the same programs used before. The considered programs are EMF models, in the form of graphs of objects which are loaded lazily. An initial load is performed to build the objects of the model but references between them are resolved on demand. Language services browse models when performing their internal logic. This process requires resolving visited references, which consist of finding the referenced elements in the model, and therefore has a cost in time. This means that language services performing simple queries on a program, such as the *completion* feature which browses few object references, are less impacted by model loading than language services that browse the whole model, such as for instance the *documentSymbol* feature which collects all named elements of a model.

Figure 11 shows the means of the measurements for the four languages, on programs of increasing size. Since models are loaded lazily, we measured the initial load times and the reference resolution times. The *load* curve represents the times to parse programs and build the corresponding model. The *resolve* curve represents the times to resolve all the references between objects in the model. The *load* and *resolve* are cumulative times that correspond to the complete model loading.

In all cases we observe a linear time for *load*, while *resolve* can be exponential according to the size of the considered

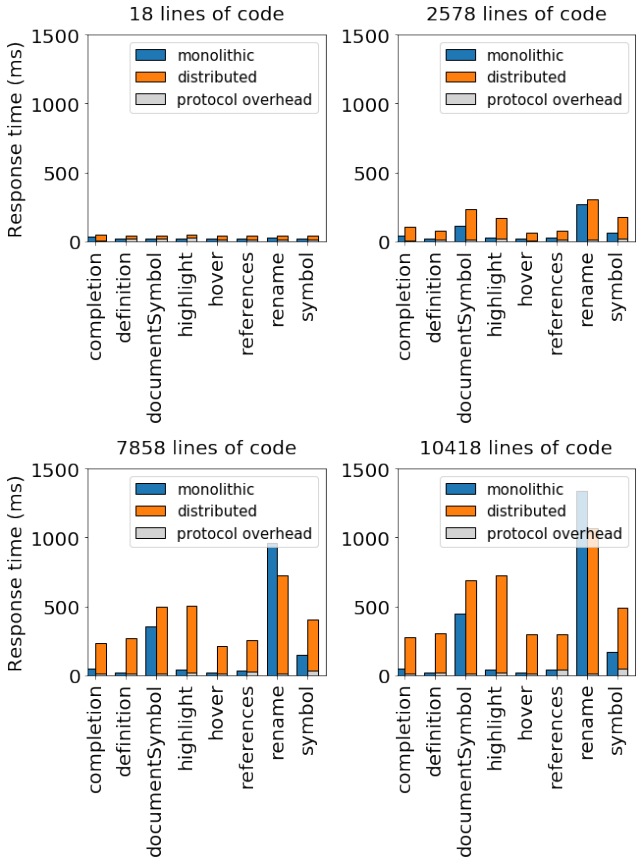

Figure 9. Comparison of response times in monolithic and distributed architectures for language services of MiniJava

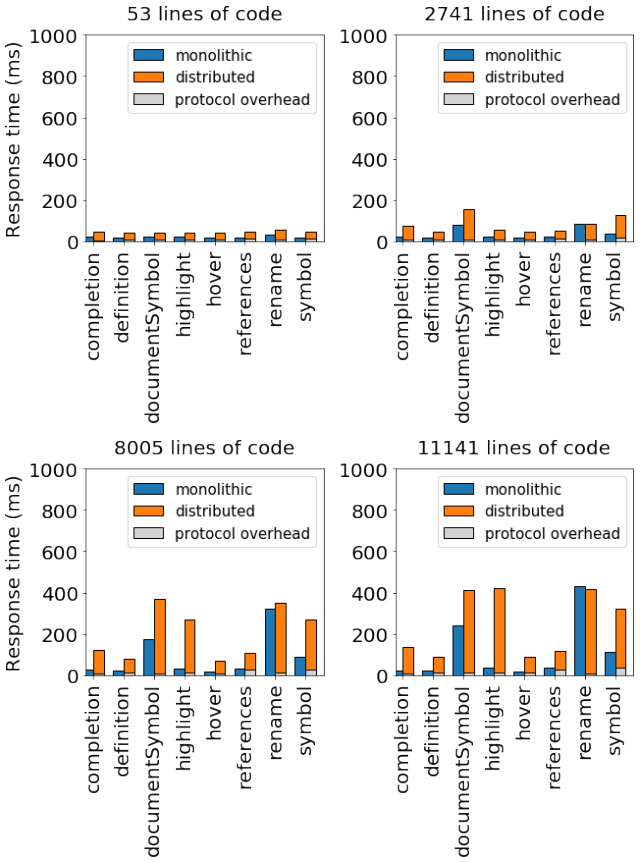

Figure 10. Comparison of response times in monolithic and distributed architectures for language services of ThingML

program and consume an important part of the overall model loading time.

To further compare the load and resolution times of model with language service response times, we measured *completion*, which is one of the fastest of our language services and *documentSymbol* which is one of the most time consuming. Figure 12 shows their response times on programs of increasing size for the monolithic and distributed architectures of the four languages. The differences of response times of *completion* and *documentSymbol* between monolithic and distributed architectures go up to 355 ms and 220 ms respectively for Logo, 347 ms and 2058 ms for NabLab, 229 ms and 242 ms for MiniJava, and 110 ms and 174 ms for ThingML.

5.3 Discussion

We modularized and distributed language services in a cluster and we compared their response times with monolithic servers implementing the same features. In this section we discuss these results to answer the research questions introduced in section 2.

5.3.1 Systematic approach to automate the modularization and individual deployment of language services

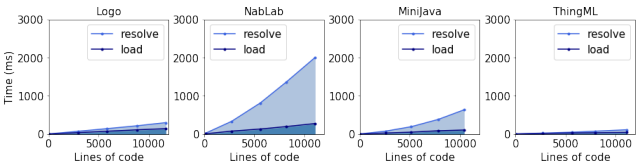

Figure 11. EMF model load and resolution times

(RQ1). We present in this paper a first generative approach to modularize all services of a language server in the form of deployment units as microservices, and a second generative approach in charge of establishing the communication between the various microservices corresponding to a given configuration of the expected IDE. The granularity of the IDE features in terms of the language services to be included in a given microservice, as well as the information on the possible dependencies between them, are given by a protocol specification taken as input of our overall approach. The microservices are stateless, and support custom and possibly dynamic adaptation of their configuration.

We experiment our approach on four representative EMF-based languages, namely NabLab, Logo, MiniJava and ThingML,

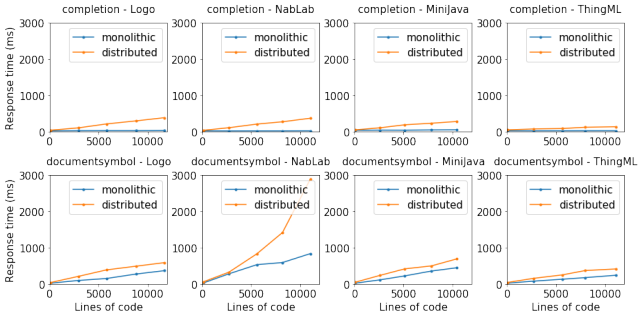

Figure 12. Comparison of services browsing few model references (completion) and many model references (documentSymbol) for the languages Logo, NabLab, MiniJava and ThingML in the monolithic and the distributed architectures

and demonstrate the ability of our approach to be applied to all of them. We generate deployment configurators for these four languages, and select language services to be deployed, distribute them on the machines of a cluster and dynamically change the deployment configuration. For all languages, the monolithic and distributed implementations of the language server are functionally equivalent.

Our evaluation shows that language services last longer in distributed architectures compared to monolithic architectures. We have identified three sources of overhead, measured in figures 7 to 12 which we discuss afterward: the modularization, the distribution, and the stateless nature of language services.

The results from Figures 7 to 10 show that the overhead due to message exchanges induced by the modularization and latencies due to the distribution (the grey parts) is only a small part of the overall overhead in the response times of the distributed features. The overhead for features *rename* and *symbol* last longer, which can be explained by the fact that these two features query the symbol index in addition to retrieving the model while the other features only send requests to retrieve the model. The most of the overall overhead comes from the orange parts, which represent the times to load models and process the language services.

The differences of response times between the monolithic architecture and the distributed architectures can be explained by the fact that the monolithic servers are stateful and the microservices are stateless. The monolithic servers load models only once at initialization and perform model validation to find errors, a process that requires traversing all elements of the model. This means that models are loaded and all their references are resolved when the internal logic of the feature is running. In the case of microservices, models are loaded at each request and references are resolved on demand while the internal logic of the feature is running. We notice in Figure 12, that the differences in response time between the monolithic and distributed curves are at least equal to the loading times of the models in Figure 11 and even close to the full resolution time of the models for NabLab. We also notice that the *completion* feature has less overhead than the *documentSymbol* feature. This difference is due to the lazy loading of the model since *completion* has to resolve few model object references whereas *documentSymbol* has to resolve all containment references of the model.

From these observations we conclude that the protocol for stateless microservices implementing IDE features introduces a small overhead corresponding to message exchanges. The differences of response times between the monolithic and distributed architectures are mostly due to the stateless nature of microservices that requires to load the model and to resolve the references lazily.

5.3.2 Distribution of the language services to improve the overall performances of the IDE (RQ2).

Our generator uses the specification of a communication protocol to generate microservices communicating by HTTP requests. Each microservice is associated with an HTTP resource, which is identified by a URL address. To send a request to a resource, a Domain Name System (DNS) must translate the destination URL into an IP address. This process abstracts the actual destination address of a request. We use HTTP to convey messages to microservices that can dynamically change their location after a reconfiguration of their deployment. Since microservices are by nature isolated from each other, HTTP communication allows the microservices to be distributed over different execution platforms.

We observed from Figure 6 that there is a real benefit on the response times to distribute some computationally intensive features (*e.g., compiler*) on powerful machines, even if they are stateless and are not co-located with the client, while others that are less demanding in terms of resources are better deployed locally to keep the best user experience (*e.g., completion* and *references*). The size of the program considered is also important on the result (cf. *rename*).

We conclude that there is an important benefit in modularizing the language services, and in distributing them in a relevant way such as we can optimize the response time of each feature individually, and improve the overall user experience of the IDE. As future work, we plan to use a learning model to estimate the pros and cons of the distribution of each feature according to the communications and the available execution platforms, and then to infer automatically the best configuration for a given context.

6 Related work

Our approach to the distribution of an IDE involves the specification of IDE features, the generation of microservices implementing IDE features and the configuration of deployments of IDE features.

Eclipse Che [4] is an online IDE using containers to the provision of IDE features. All the IDE features for a language are embedded in a container. Eclipse Che is only configurable at the granularity of a language and doesn't allow to select the deployment location of the containers. Our configurator operates at a granularity of the IDE feature and allows to distribute them on different locations.

Monto [8] is an approach to decouple IDE from IDE features. The authors manually implemented IDE features as microservices and specified IDE agnostics intermediate representations (IR) for four IDE features to communicate with them. Our approach is to generate microservices from specifications. Monto does not cover the concern of IDE configuration.

DISTIL [1] is a DSL to describe cloud-based Model-Driven Engineering (MDE) services (*e.g.,* model transformation, model queries, ...) that are close to what we call IDE features. This DSL allows to generate persistence service, REST services and HTML front-end. MDE service descriptions can declare input and output and a validator checks correctness of connected inputs and outputs based on their types. DISTIL allows to describe the data structure of elements stored in the persistence service. Our approach does not provide a generator for the front-end and we do not generate persistence service. Our approach allows to specify dependencies of IDE features and provide a feature model to ensure a safe configuration.

Jolie [6] is a programming language to implement microservices and specify microservice architectures. It provides abstractions for microservices coordination to be used in implementations. We provide a language to describe the dependencies between language services, and based on such descriptions, we automatically generate for each microservice the source code needed to call the dependencies. However, we rely on the language designer to complete microservice implementations by calling the dependencies in the right order. Our language could benefit from such coordination abstractions to go further in the automatic generation of microservices.

MicroBuilder [12] is an approach to generate microservices with REST API from a specification of a microservice architecture. Their specification uses REST concepts, such as HTTP verbs and does not describe dependencies between microservices. Our specification uses a domain specific model to define such information to not be tied to one technology. The description of dependencies between IDE features allows us to generate source code to call other microservices.

In [3], authors propose a metamodel to specify microservice architectures to generate synthetic microservices for benchmarking. Their metamodel allows specifying microservices with REST API and their dependencies, which is close to our protocol metamodel, and allows specifying deployment configuration composed of instances of the microservices. Our approach does not allow to describe microservice instances, as we let the cluster manage their replication automatically.

In [11], authors report experience on manually splitting Sharelatex (a web editor for Latex documents) into microservices, using a hybrid core/edge deployment of this application. They identified criteria to split the state of microservices for replication and criteria to decide on the deployment location of the microservices. Our approach does not address the concern of microservice replication and we do not describe deployment location criteria for IDE features, but we provide a method to automatically generate microservices and support reconfiguration of their deployment.

Our approach proposes an end to end solution, from a language specification to the IDE deployment, to configure and distribute IDEs by automatically generate microservices for IDE features and a configurator for their individual deployment. Our experiments demonstrate the benefits of such an individual deployment.

7 Conclusion and Perspectives

This paper discusses the modularization of language services to support their individual deployment and dynamic adaptation within an IDE. We propose a generative approach to automatically obtain microservices implementing IDE features from a language specification, complemented with a feature model that drives the safe configuration and automate the deployment of IDE features. We study the impact on performances when distributing the language services across the available execution platforms. We evaluate our approach on four EMF-based languages and demonstrate the benefit of a custom distribution of the various language services. Our experiments highlight the usefulness of distributing computationally intensive language services. Indeed, the use of distant application servers improves the performances of some language services (*e.g., rename* and *compile*), thereby compensating the overhead due to the distribution. Experiments also show the benefits of retaining locally other features that should be reactive and less demanding in resources.

These experiments highlight three important research locks that we identify as directions to be investigated by the community. Firstly, the load of models implied by stateless microservices is costly. Hence, choosing its location and proposing effective caching or replication techniques decided on the basis of the information contained in the language specification is an area to be explored, especially when we integrate collaborative editing scenarios. These scenarios were outside the scope of this paper. Stateless microservices open the possibility to share sets of load-balanced instances of language services between language users. It could be interesting to study the scalability and the resource consumption of stateless microservices and compare it with a monolithic architecture in a multi-users scenario. We demonstrated in this paper it exists a trade-off between stateless and stateful

language services but further investigations are needed to determine which nature is better for each language service.

Secondly, the availability of the WebAssembly[24] technology creates a new technical environment that facilitates the development of modules that can run both on the client side and on the server side in an efficient manner. Reflecting on the granularity of these services and the technology stack that allows one to embrace as many programming languages as possible in designing these services while allowing both client-side and server-side execution remains a great challenge.

Finally, the construction of the adaptation logic adapted to the IDE operation to improve the user experience and follow the evolution of its needs by automatically migrating or reconfiguring a service is still a widely open research field.

These three locks will require further investigation to characterize language services. For example, studying their frequency of use and how they access the model will help determine whether they should be stateless or stateful and whether it is better to query part of the model on demand rather than obtaining all of it. Studying their memory or CPU needs will help determine the best deployment. The application of static analysis techniques on the source code of monolithic software implementing language services is another direction that could be explored to automatically extract protocol specifications and determine which service can be asynchronous.

If these three challenges are widely addressed by distributed computing communities, web engineering communities or adaptive systems communities, the specific case of IDEs brings a certain amount of information that allows specific reflections on each of these topics.

References

[1] Carlos Carrascal-Manzanares, Jesús Sánchez Cuadrado, and Juan de Lara. 2015. Building MDE cloud services with DISTIL. In *International Conference on Model Driven Engineering Languages and Systems (Model-Driven Engineering on and for the Cloud)*, Vol. 1563. CEUR Workshop Proceedings, Ottawa, Canada, 19–24. https://hal.archives-ouvertes.fr/hal-01761670

[2] Abhijit Dubey and Dilip Wagle. 2007. Delivering software as a service. *The McKinsey Quarterly* 6, 2007 (2007), 2007.

[3] Thomas F. Düllmann and André van Hoorn. 2017. Model-Driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion)*. Association for Computing Machinery, New York, NY, USA, 171–172. https://doi.org/10.1145/3053600.3053627

[4] Eclipse Foundation. 2020. Eclipse Che | Eclipse Next-Generation IDE for developer teams. https://www.eclipse.org/che/ [Online; accessed 25-February-2020].

[5] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 307–309. https://doi.org/10.1145/1869542.1869625

[6] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. 2017. Microservices: a Language-based Approach. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer, https://hal.inria.fr/hal-01635817. https://hal.inria.fr/hal-01635817

[7] Jay Heiser and John Santoro. 2019. Hype cycle for software as a service.

[8] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE Portability Problem and Its Solution in Monto. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/2997364.2997368

[9] Andres J. Ramirez and Betty H. C. Cheng. 2010. Design Patterns for Developing Dynamically Adaptive Systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*. Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/1808984.1808990

[10] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional, https://www.amazon.fr/Budinsky-Eclips-Modeli-Framewrk-_p2/dp/0321331885.

[11] Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. 2018. ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-Based Application. In *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets (MECC'18)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3286685.3286687

[12] Branko Terzic, Vladimir Dimitrieski, Slavica Kordic, Gordana Milosavljevic, and Ivan Lukovic. 2018. Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures. *ENTERPRISE INFORMATION SYSTEMS* 12, 8-9 (2018), 1034–1057.

[13] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[24] https://webassembly.org/