A Case for a New IT Ecosystem: On-The-Fly Computing

Holger Karl, Dennis Kundisch, Friedhelm Meyer auf der Heide, and Heike Wehrheim

I. Introduction

Are you an IT developer who recently spent days on searching for appropriate libraries reusable for your new application? Or are you a knowledgable IT user who wants to build a web application for your sports team, but gave up soon? Then you are one of many who have experienced the complexity of development and deployment in today's IT world: Despite the existence of so many pre-fabricated components, frameworks, cloud providers, etc., building IT systems still remains a major challenge. When *developing* applications, a developer should be familiar with a vast range of libraries, frameworks and environments: Application libraries constituting user frontends running directly on smartphones differ from frontends running inside a web browser. Application components that implement an application's business logic might run inside a web framework on a server, requiring access to different kinds of databases or event processing frameworks. A backend application component might do complex machine-learning tasks, which require yet another set of frameworks. When *deploying* an application, a similar embarrassment of richness is encountered: Preparing for deploying on smartphones, web browsers, cloud environments or bare metal setups is all very different, each with its own unique set of challenges. Knowing all this likely overtaxes even an ambitious developer. Today, this results in spreading such development and deployment tasks over different team members with their own specialisation. Nevertheless, not even highly competent IT personnel can easy succeed in developing and deploying a nontrivial application that comprise a multitude of different components running on different platforms (from frontend to backend).

Current industry trends such as *DevOps* strive to keep development and deployment tasks tightly integrated. This, however, only partially addresses the underlying complexity of either of these two tasks. But would it not be desirable to simplify these tasks in the first place, enabling one person – maybe even a non-expert – to deal with all these tasks?

Today's approaches to the development and deployment of complex IT applications are not up to this challenge. To put such an approach into effect, the selection of frameworks, choice of suitable libraries, acquisition of components, generation of code, and production of executable and deployable artefacts ("software") all must be much more automated than what is doable today. With proper automation, it should even become possible to generate, deploy, and execute software on suitable hardware on very short term, possibly even *on-the-fly* when the need for a particular, new piece of software arises.

We stipulate that this is a highly relevant challenge for research in many branches of computer science and adjacent disciplines such as information systems, business administration and economics. This challenge is clearly formidable yet it also does not appear to be hopeless. In this article, we analyse which pieces are in place and where additional, often significant research efforts are necessary. Specifically, we illustrate which roles and use cases we foresee for such *on-the-fly computing* (OTF computing, for short), using examples from

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901)

the conventional to the forward-looking (Section II). From these examples, we can identify challenges for software engineering necessary for OTF computing, producing components that can be handled more automatically than the ones feasible today (Section III). Such automatically treatable software components present both opportunities and challenges to the infrastructure for deploying and executing these components; we discuss which of today's approaches to infrastructure are suitable and which need to be extended (Section IV). Finally, we observe that both aspects – software engineering and "infrastructure engineering" – are mere facets of the overall engineering problem for OTF computing. There, components (e.g., software libraries), execution environments (e.g., cloud systems), and activities (e.g., development of a needed software component) will be frequently traded. We discuss how this trading can be efficiently organized in a marketplace and how OTF computing marketplaces have to be engineered as well (Section V).

II. Roles and use cases

We use three example use cases to illustrate the concept of OTF computing. This will introduce a number of **roles** (indicated in **boldface**) as well as some `concepts` and `artefacts` and relationships between them (indicated by `typewriter font`).

A. Example use cases

1) UC 1: Conventional web applications: Let us consider a typical web `application`, geared to serve many **users** – think of an online map application or a hotel booking website. In this use case, the idea for such an application was conceived and implemented by an individual or company; they requested the creation of this web application in a more or less implicit form (as today, OTF computing is not explicitly developed). Such a **requester** would (today) specify that its application consists of a couple of `components`, such as a web framework or a database. These components are made available from different sources, e.g., open-source initiatives or companies – they all take on the role of a **component provider**. In addition, code that represents the actual application semantics had to be developed; this forms yet another component to be run, e.g., as code inside a web framework.

Once all components are available, they can be run either as a single `service` or as a collection of interacting microservices. In case some of the components are not available as (source or binary) code, they might still be available as a service themselves. One example is a payment service made available via a REST interface by a **service provider** – Paypal is the canonical example here; mapping and weather forecast services are other typical examples. In summary, an application itself is a service, composed of other services that are either already running and accessed or are started as an inherent part of the application.

Such an application is typically executed on different `infrastructures`, such as users' smartphones, and on servers in a cloud system. These infrastructures are made available, explicitly or implicitly, by **infrastructure providers**: say, an Amazon web service for the web servers and the users when running frontend code in their web browser.

In this simple, well-known use case, there are two steps involved that, today, are highly labour- and knowledge-intensive: 1) the composition of an application out of simpler components, along with additional code for application semantics (today typically provided by the **developer**); 2) identifying options for executing an application's components (or accessing constituting services) on suitable infrastructures. In the following sections we shall investigate how OTF computing would make these steps more efficient. But before, let us consider two more complex use cases.

2) UC 2: Big-data applications in backend systems: Consider a machine-learning application analyzing large amounts of data in a commercial context where data scientists assist an expert from an application domain. Often, such applications change frequently and are composed of more or less simple components (e.g., database access, data preprocessing, graph extraction, clustering, various learning schemes, classification, or regression algorithms). Such machine-learning libraries exist [15] and their components can be combined, today albeit with nontrivial manual effort. In OTF computing, this effort should be substantially reduced.

In this use case, the domain expert would be the **user**, the data scientist assumes the role of the **requester** on behalf of the user. `Components` such as Spark [30] are provided by corresponding **component providers** (here, the Apache consortium); `services` such as data visualization (e.g., Plotly) are provided by the corresponding **service provider** (plot.ly). Some of the `infrastructure` to run data analysis could be hosted by the data scientist's company, taking care of the **infrastructure provider** role as well.

The data scientist provides additional expertise here, which is a key contribution to the application: which components should be meaningfully composed into a big-data analysis `application`? It is a typical activity: composing components and services together to form a new, useful component (or application), based on semantic, domain-specific understanding of the problem at hand. Moreover, it could be necessary to orchestrate the execution of an application on one or multiple infrastructures, catering to the particular needs of the various services.

In today's software engineering approaches, this role is not explicitly visible; we do believe, however, that it is a key aspect for future development approaches. We hence highlight it explicitly and christen it **broker** to point out its role between different other roles: The broker needs to understand requesters (and, implicitly, users) as well as the offer of component/service providers and infrastructure providers. It needs to be able to create a new component out of existing pieces plus, potentially, generating additional glue code between those pieces. This is a considerable challenge to be pursued.

3) UC 3: User-triggered smartphone application generation: In the most forward-looking application, the role of the **broker** becomes even more important. Let us imagine that a broker is able to understand the needs of a user very well and can create an application to be executed on the user's smartphone directly, on-the-fly, as a user expressed his or her idea. This will require considerable understanding of ill-expressed, informal requests. It is far beyond today's capabilities in natural language understanding and software synthesis but serves as an interesting target scenario.

In particular, in this scenario, another issue becomes apparent: Even assuming that a broker were capable of all this semantic understanding, it is not obvious where such a broker should search for components or services to be used in generating such an application. Nor is it apparent where such services could be offered in the first place. Hence, we identify a last role necessary to implement OTF computing: A **market provider** that creates a marketplace, operates it, and mandates and polices rules of who is allowed to offer and request what (components, services, infrastructure, . . .) under which kinds of licences, etc. Again, this role is only in place for some kinds of IT systems today; for example, Apple or Google are market providers by means of their app stores. But these stores fall short of supporting the search functionality of components. Today, this role exists only in a rudimentary, highly manual form; it is shared over code websites such as GitHub, cloud providers such as Amazon or cloud consolidators such as HashiCorp. We expect an OTF computing ecosystem to be able to support single or multiple market providers and brokers, having access to automatically searchable component and service repositories.

B. *Roles in an OTF world*

In summary, OTF computing comprises a number of familiar roles; it also points out the significance of a couple of roles that are present today, but are only filled implicitly or with a limited scope in comparison to an OTF implementation. First and most important among those is the **broker** role. The first will be crucial to create new components, services and applications and orchestrates the development of newly required functionalities. Second, the **market provider** will provide the organisational, economic, contractual and possibly legal framework in which an OTF economy can develop and flourish.

The key relationship of these roles is summarised in Figure 1. It shows only the most important interactions between roles.

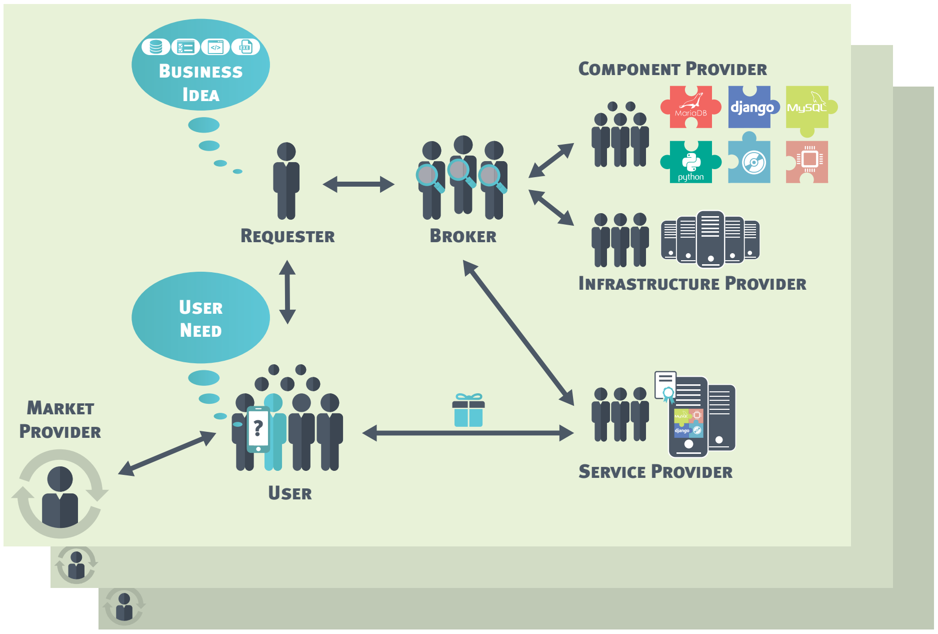

Fig. 1. Roles and their relationships in OTF computing

III. Software Engineering in OTF

OTF computing poses three key challenges to software engineering related to the development of applications and the interaction of requester and user with the OTF ecosystem as a whole.

Description	How to tell the broker what is to be generated or executed? And, equally important, how to describe what is available (components, services, infrastructure) and thus usable by the broker? The options range from natural language descriptions to full formal specifications.
Configuration	How to assemble which components and services? Where to deploy them? On an abstract level, this is a planning or optimization problem; for the final execution, this is a question of interfaces, platforms, technology standards and hardware.
Quality assurance	How to ensure high quality? Current software engineering methods for quality assurance (e.g., testing, analysis, monitoring, certification) can be leveraged, but need to be adapted to the OTF computing context.

The challenges and hence the methods to tackle them are closely interconnected. We discuss them in more detail and explain, in particular, which roles face which challenge.

A. Description

Within the OTF ecosystem, three roles need *languages* to describe requirements (on applications to be built and on single components or services to be found in the market) or guarantees (of the specific infrastructure provided): The first is the requester, e.g., an employee in a company or just an individual. Preferably, she would not have to learn a specific language for writing requirements. Rather, she would like to fill in a form or directly write her requirements in her own mother tongue. The second role is the broker. For its task, it needs languages for directly asking providers or querying the market for a specific entity. The third sort of role is taken by all providers (of services, components or infrastructure). They need to precisely formulate the guarantees provided by their entities. For all of these tasks, domain-specific languages could be envisaged, but also general-purpose languages adequate for specifying all aspects of services, components and infrastructure could be designed. The difficulty we see here is not in the existence or design of languages, but in the willingness to employ and standardize such languages in a broad scope.

With respect to formulation of natural language requirements, already today we see the rise of systems communicating with humans in natural language [24], [13] (virtual assistants such as Apple's Siri or Amazon's Alexa). They "understand" questions and can provide answers. Such systems are, however, still far away from the demands of OTF computing: 1) They merely give answers when the knowledge is already at their disposal (e.g., in a database). These assistants can thus be seen as some advanced way of querying databases. In OTF computing, requesters will, however, describe requirements of compositions that do not exist. Still, the request needs to be "understood" to an extent which enables configuration of an appropriate composition. 2) They have limited possibilities for enquiries in case of non-understanding. Enquiries are made when the vocabulary used in the query is unknown or when there are acoustic reasons for non-understanding. OTF computing needs *content-wise enquiries* to close gaps in the knowledge, learn new knowledge and suggest alternatives when no direct answer is available [9].

With respect to formal or semi-formal languages for describing requirements or guarantees, there are numerous proposals in the area of component description. Formal methods in general or techniques tailored towards software, such as Design-by-Contract, allow formalizing requirements and guarantees in various degrees of precision. As there is no standard description language, the challenge lies in *matching* [20] differently formulated requests and guarantees against each other.

Research required: To achieve true OTF computing systems, novel techniques for natural language processing, building dialogs with users and learning from past dialogs are required. On the specification language side, our prediction is that no universal language will emerge, but rather many domain-specific languages. The key requirement in all these areas is, however, not language design but willingness to converge, standardize and use a small set of languages.

B. Configuration

In OTF computing, configuration refers to assembling an application that meets the requester's requirements. It is the task of the broker. Current approaches to automated software and service composition [16] typically take on one of the following forms: *Template*-based approaches assume the structure of the service composition to be given a priori and configuration consists of instantiating placeholders, setting parameters, or customizing variants (e.g. [3]). These approaches are applied in web service composition or for software product lines. They often amount to solving an optimization problem. *Free* configuration builds on no such assumptions. The construction of the overall structure is part of the configuration process itself. Such approaches require formal specifications of both the requirements and the services (in propositional or first-order logic, e.g. [10]). Free configuration approaches rely on AI *planning* in one form or another.

While the drawback of template-based approaches is the necessity of knowing the template beforehand, the difficulty for free configuration lies in requiring formally specified services. Again, it has to be stated that such specifications are often not available. As another drawback of free configurations, planning often builds only sequential compositions of services/components: i.e., it only finds a very limited structure. As a consequence, neither template nor free techniques are applied on a wide scale today.

An exception are domain-specific configurations, an example being automatic assemblance of machine learning applications (AutoML, e.g. [28]). Domain-specific approaches can typically go beyond general approaches by leveraging expert knowledge on the application domain and its structure and thus achieve a higher degree of automation.

Research required: The template-based configuration approach needs more general ways of describing templates; free configuration needs techniques for building more complex structures. There are, however, also general limits to any progress in this area, due to composition (depending on the exact setting) being an NP-hard or even undecidable problem. The future might thus lay in numerous domain-specific techniques where generalization across application domains can only be achieved by setting up a framework for configuration.

C. Quality assurance

Quality assurance refers to achieving high-quality applications, with respect to functional as well as non-functional properties. Three roles are responsible for or may influence the quality of a service composition. The first is the service or component provider, whose interest may, however, just be in achieving a high price for his service/component or high sales figures (Section V). The second role is the broker system. As it has only limited influence on the entities plugged into an application (it selects components/services, but cannot influence their internal operation), it either needs to be supplied with techniques for checking service/component properties or has to rely on user reviews or ratings. The key challenge for the first case lays in the “on-the-fly” nature of OTF computing: While there are numerous software analysis and testing methods for ensuring the safety and security of services, these are typically not aligned to an “on-the-fly” usage. Notable exceptions are specific certification techniques that target safety properties [17], [25], [11]. Software certification attaches correctness *proofs* to software and builds proof checkers inspecting validity of proofs. Such approaches already work towards the concept of “on-the-fly” quality checking as proof checking is typically faster than proof construction. The key challenge for the second setting (the broker relying on user ratings) is the fact that applications are rated by users as a whole; in fact, the *application execution* is typically rated. The broker cannot easily see which parts of an application are the cause of a bad or good rating; it might rest in some application component or in an unsuitable infrastructure. This question is known as the *disaggregation* problem (which is discussed in more detail in Section V).

The third role is the infrastructure provider responsible for meeting service-level agreements; this is discussed in Section IV.

Research required: A truly “on-the-fly” quality assurance requires major improvements in the speed of software analysis techniques, whether they involve certification or not. For certification, more analysis approaches need to be able to construct “proofs” and be supplied with proof checkers. Disaggregation techniques for user ratings have to be developed and put into practice.

IV. IT INFRASTRUCTURE FOR OTF

Obviously, an application becomes useful only once it is executed. To execute it, some form of execution machinery is necessary – we summarised this under the umbrella term of *infrastructure*. The question here is to deal with the infrastructure as such as well as with the *mapping* (or *scheduling*) of components to specific instances of the infrastructure for execution.

For infrastructures and for executing components on top of them, OTF computing provides challenges in the same three categories (Description, Configuration, and Quality Assurance) as discussed in Section III. We discuss these challenges in turn.

A. Description

As already pointed out in Section III, a description of the capabilities of an infrastructure is an essential precondition. Foundations for that are currently being laid by data center description languages such as the Data Center Markup Language (DCML, dcml.org), hopefully resulting in standard interfaces for data centers in the sense of the Software-Defined Data Center concept. Obviously, this goes along with virtualization of the infrastructure, putting data centers under the control of cluster management systems (e.g., Mesos, mesos.apache.org) along with suitable interfaces. The idiosyncrasies of different infrastructure providers – today, usually synonymous with cloud providers – can be hidden by abstract layers such as the one provided by Terraform, terraform.io. Nonetheless, standardized description languages with clear semantics are still largely absent.

More generally, the notion of *infrastructure as code* is currently gaining momentum: It allows to describe an application's specific needs and to provision it on-the-fly, when an application starts up. This includes, for example, provisioning a suitable number of (usually virtual) servers along with a required network and storage setup.

While infrastructure as code does constitute an excellent basis for OTF computing, it is nevertheless insufficient. For example, the current focus of data-center oriented descriptions is understandable given the economic drive of cloud computing and large setups. But it does not do justice to the needs of OTF computing where we foresee a much wider range, and much a richer versatility of infrastructure. In practice, there is currently no consistent approach to describe execution options for a given application that might be run on either a smartphone or a backend system (or where functionality might even be dynamically distributed between these two execution environments). As another example, an application might be runnable on a standard CPU architecture or on GPGPUs or FPGAs (with different binaries included in the application description, of course). This requires a description of the corresponding capabilities of real systems, bypassing many virtualization layers that explicitly try to abstract away such differences. Also, for large applications with a geographically distributed user base, a single data center might not be the right solution – distributed cloud computing might rather be preferable – this necessitates knowledge about multiple data centers and their interconnectivity.

Required research: To address these scenarios and needs, our description techniques need to support *infrastructure polymorphism*: vastly heterogenoues and vastly distributed infrastructures. While a lot of such information is available in more or less implicit form (e.g., geographic distribution of data centers), none of that is available in a formalized, machine-readable, standardized way.

This leads to the second issue: which infrastructures are at hand? Today, the choice where to execute a service is extremely limited as no consistent description is available, nor is there any standardised way of finding such information. We need an *infrastructure discovery* mechanism that can provide opportunities from the small to the very big.

The ensuing question is what to include in these descriptions. Obviously, quantitative descriptions of capabilities are relevant, but also cost information, offered SLAs, etc. Without that, mapping components to infrastructure would boil down to guesswork as is done today when little guidance exists which cloud provider to use. Ideas in this context are legion, usually tracing back to the rich literature on SLAs, but also on grid computing. OTF computing goes beyond that, however, in requiring a good understanding of load: Some types of OTF applications will be characterized by large user populations, distributed world-wide, with possibly widely differing requests (think a next-generation video-on-demand streaming provider).

Required research: To address these needs, existing formalisms (e.g., ETSI's NFV description formalisms [8]) need to be formalized towards *geographically distributed load profiles*, constituting a rich research field, in particular, when incorporating time-varying load predictions.

Obviously, all these description approaches need proper standardization and sufficient buy-in to get an OTF ecosystem started. This is discussed in more detail in Section V.

B. Configuration

With proper, qualitative and quantitative descriptions of both infrastructure and applications available, the task at hand is configuration. In this context, that means figuring out which particular version of an application (or a service) to deploy, possibly choosing not only which but also which kind of infrastructure, where to serve which particular demands, and how many resources to assign. Typically, this boils down to rather complex optimization problems, often to be solved with real-time constraints or in an online setting.

For relatively simple applications, the closest research fields here are Distributed Cloud Computing and Network Function Virtualization. Both concern themselves with such placement and automatic scaling problems, along with lifecycle management of such composed applications. This work is typically focused on wide-area infrastructures and ignores internal organization of "compute nodes", which might in turn be entire data centers. Inside data centers, on the other hand, very little explicit information about an application is taken into account. A first example in this direction, focused on data-parallel applications such as Map/Reduce jobs, is the notion of a coflow [6], where flows of a single application are synchronized. However, there is very little support available for scheduling such applications according to complex SLA requirements.

Required research: There is a large range of research that is needed here. Even assuming that all information about infrastructure would be available, there is still a wide range of resulting optimization issues. We do need fast yet reasonably precise heuristics or, ideally, approximations for complex scheduling, scaling, and placement problems, especially across heterogeneous infrastructures for versatile services, capable of running on different execution environments. Inside data centers, this problem is aggravated by the many system layers and the resource competitions among applications; on the other hand, it is simplified by the complete control over all these layers. In wide-area networks, many simplifications that are acceptable in data centers no longer work (e.g., equal delay along all paths), but application structure is typically much simpler than in complex data-parallel applications.

In total, configuring complex applications and multi-tiered, heterogeneous infrastructure will stay a considerable challenge for the foreseeable future.

C. Quality assurance

With respect to execution, OTF computing shares many of the challenges of generic cloud computing: Policing SLAs, ensuring privacy of execution, and making cost claims transparent are all issues shared across many approaches. The OTF computing case, however, has some additional challenges: attributing any violations of an SLA to the right instance. In both conventional and OTF computing, a data center might have failed to live up to its promises, or the software was not up to the task.

Required research: Quality assurance for distributed computing has a long history. The composed nature of typical OTF software and the role of the broker add another level of complexity here. As the software engineering Section III already outlined, a mechanism for *disaggregation of responsibility* is needed.

V. Markets

A marketplace can be understood as a set of rules to organize interactions and, ultimately, transactions between involved parties. To refer to the Appstore example mentioned above, the Appstore provider (i.e., the market provider) must define who is allowed to participate, who may offer an app, if there is a commission to be paid to the market provider for every sold app, and so on. A successful marketplace is organized in a way that the marketplace grows and increases the shareholder value of the owner of the marketplace (here: the market provider).

Apparently, even if all previously described technological challenges (see Sections III and IV) were mastered, an OTF market place's chances of success are not only determined by technical aspects; they also hinge on these rules from a business perspective and on applying legal constraints. Hence, there is a need to structurally think about and develop such a marketplace, otherwise the likelihood of wasting invested resources is high. This includes, amongst others, setting the right incentives for all the different groups of participants to join and haunt the marketplace as well as aligning these rules on the business layer with the implementation of these rules on the application and infrastructure layer of the marketplace itself. Hence, it not only has to be defined, who may offer something in the marketplace, but the actual processes also have to implemented to materialize this offer in the marketplace; and these implemented processes have to run on some suitable infrastructure. OTF computing poses challenges not just for software engineers or researchers focusing on the necessary infrastructure but also for the agents setting the rules of such a socio-technical system. Key challenges for OTF computing relate to the information asymmetry, the multi-sidedness, and the enterprise architecture.

Information Asymmetry	How to avoid an OTF marketplace from failing from a business perspective given the manifold information asymmetries present between the many market participants with respect to individual behavior and actual quality of services and applications?
Multi-sidedness	How to best exploit the strong cross-side network effects in an OTF market place? How to overcome the mutual baiting problem?
Architecture	How to best support the build-up of an OTF marketplace spanning all three layers: business, application, and infrastructure? And how to cope with the fundamental dynamics of an OTF marketplace and its necessary iterative development?

We will discuss these challenges in turn.

A. Information Asymmetry

Developing a "good" set of rules is not easy, due to the presence of information asymmetries between the involved parties. In comparison to the requester, let alone the user, a broker in an OTF marketplace, for instance, has a lot more information about the available services and their respective quality that may be used to compose an application. Ever since the seminal contribution by Akerlof [1] it is established in the literature that in the presence of information asymmetry between buyers and sellers, an adverse selection problem may emerge that drives higher quality out of the market and may even lead to a market collapse. The traded objects in an OTF market are unique, user-specific – or, strictly speaking, requester-specific – service compositions. Hence, they are dominated by experience [18] and credence attributes

[7]. Experience attributes can be known only after using a product/service, while credence attributes cannot be evaluated by a consumer even after consumption but have a perceived value. For decades a doctor's visit has been a typical example of a service that is dominated by credence attributes. Today, many complex digital service compositions, for instance in the area of machine learning, share this characteristic. Obviously, this makes a quality inference extremely difficult ex ante for the user. At the same time, the broker and all further participants on the supply side of the marketplace have a hard time figuring out the user's actual willingness to pay if it is not already provided in the request.

Two basic means are used to mitigate such information asymmetries [22]: signaling (e.g., advertising, granting warranties) and screening (e.g., performing market research, imitating reference customers). The literature already provides valuable insights into the mechanisms to mitigate information asymmetries, for instance, by soliciting electronic word-of-mouth communication in the form of consumer ratings [4] and learning about the product quality from these ratings [12], [31]. Still, the distinct features of OTF marketplaces (i.e., unique service compositions, a vast amount of possible service compositions for a specific request, and low or even negligible marginal costs of producing and distributing a service composition) require further research in this area.

Research required: Complementing the signaling capacity of software certification (see Section III), new efficient and effective signaling and screening mechanisms on the business layer have to be developed that mitigate existing information asymmetries on an OTF marketplace. It seems promising to expand research on electronic word-of-mouth communication in general and consumer ratings in particular, as they have become the de facto standard of reputation systems on virtually any platform market.

B. Multi-Sidedness

Developing a "good" set of rules is not easy, because an OTF marketplace is a multi-sided platform market [19]. Multi-sided platform markets are economic platforms that have two or more distinct groups of participants (see Figure 1) that provide each other with network benefits (referred to as two-sided markets if there are exactly two groups of participants). Network effects can emerge on one side of the platform and across sides. Cross-side network effects give rise to the chicken-and-egg dilemma (also referred to as the mutual baiting problem) of early-stage, multi-sided platform markets [26]. This dilemma describes the need for a critical number of participants (e.g., service providers) on one side of the marketplace to attract participants on another side (e.g., requesters); however, service providers will adopt the marketplace and only invest if they observe a sufficient number of requesters on the other side – or at least expect them to join. Once a multi-sided platform market reaches the critical user mass on each side, network externalities stimulate self-reinforced platform growth.

Network effects put a much stronger focus on dynamic aspects of economic interactions. Hence, it is not enough to design a "good" set of rules; these rules have to be modified on an ongoing basis by the platform owner (in our case the market provider), depending on the market environment. Research has just begun to understand the dynamics and consequences of the presence of strong network effects on aspects such as price setting, platform growth and competition of multi-sided platform markets. A couple of platform launch strategies have already been proposed [19], [26], but these strategies so far focus almost exclusively on two-sided settings. An OTF marketplace, in contrast, may have up to six distinct groups of participants (see Figure 1).

Research required: The scope of the analysis of two-sided platform markets has to be extended to cover multi-sided platform markets with three and more distinct groups of participants. Analytical modeling will be one method of choice here and the necessary extensions are straightforward, but the models are getting intractable rather quickly and interpretation of closed-form results is getting increasingly difficult. Simulating the establishment of an OTF marketplace that employs some launching strategy (or a combination of several launching strategies) therefore also seems to be a promising approach.

C. Architecture

To build up an OTF marketplace, system architects – the person "responsible for the whole-system view" [14] – apply an integrated view spanning three layers: business, application and infrastructure. Ever since the seminal contribution by John Zachman in 1987 [29], enterprise architectures have been in the focus of research and business practice alike. Despite recent advances, surprisingly little design knowledge is available when it comes to the design of digital marketplaces instead of enterprises. In fact this makes a huge difference, as the focus of enterprises is more on organizing the production of the service or good within the supply chain, while the focus of marketplaces is on facilitating economic interaction between market participants. Hence, the incentives of individuals or firms to participate in the marketplace have to be taken into account when designing the marketplace on a much broader scale. In addition, one should note that there are many flavors or variants of an OTF marketplace. An OTF marketplace could take the form of a public marketplace across firms or an "in-house" marketplace within a firm, as a domain-specific (e.g., machine learning applications, office applications) or a domain-independent marketplace, as a B2B or B2C marketplace, as a marketplace with competing brokers or just one broker, and so on. Consequently, for each variant a specific set of rules as part of the business architecture has to be found that make this OTF marketplace successful. And these rules then have to be translated into requirements for the application and the infrastructure layers.

Designing the business architecture involves designing the business strategy, governance, organization, and business processes. When it comes to the governance, organization, and especially business processes, there is both ample modeling support and design knowledge available, including, for instance, reference process libraries (e.g., [2], [23]). Developing a business strategy and – as a pivotal part of that – a viable business model are, however, clearly underdeveloped in terms of design knowledge, process, tool support, and modeling support. A business model describes the mechanisms of how a firm creates, delivers and captures value [27], and as such can be understood as a detailed description of a firm's strategy [5]. Business models are important because firm performance depends not only on the characteristics of the products or services a firm offers, but also on the business model employed for commercializing these products. Akin to process modeling, creating and innovating a business model is a creative and collaborative process. As with many creative processes, the outcome of this process cannot be definitely judged with respect to the quality upfront (i.e., "Will this business model be successful?"), it has to be tested and typically adapted in an iterative fashion (applying, for instance, the so-called "build-measure-learn feedback loop" [21]). Due to the existing information asymmetries and the multi-sidedness of an OTF market, the iterative and dynamic fashion of an OTF business model development is especially pronounced. Integrated multi-level modeling spanning the business, application and infrastructure layer of an OTF market is already a challenge if the business model is rather stable. The Open Group Architecture Framework (TOGAF) could be used as a starting point for this endeavor as it is the de facto industry standard for designing, planning, implementing, and governing an enterprise information technology architecture.

Research required: A meta-model for OTF markets – or, on an even broader scale, for IT service markets – has to be developed including a domain-specific language, a variability model to account for the different variants of an OTF marketplace, a method to build up an

OTF marketplace and a governance framework comprising conformance checks. Such a meta-model would help in designing a specific OTF marketplace much faster and in higher quality. Special emphasis should be put on: (a) the method that should allow for a hypothesis-driven dynamic development of the market and (b) the impact of these dynamics on the application and infrastructure layers, both in terms of modeling (one might even refer to it as on-the-fly modeling) and (on-the-fly) implementation.

VI. Conclusions

Even in today's IT world with numerous sophisticated frameworks, libraries and environments, developing and deploying IT applications remains a challenge. This article has advocated *on-the-fly computing* as a novel IT ecosystem, forseeing a great deal of *automization* in configuration and deployment as the key driver towards solving this challenge. This article has furthermore identified the *organization of markets* as the core ingredient of successful OTF computing, to give incentives to all stakeholders in OTF computing and to achieve alignment of technical and business needs. Though a lot of research is still required for this vision, we already see aspects of OTF computing come into existence today, within specific application areas. This makes us confident that OTF computing is not a mere research idea, but a vision which is to become reality soon.

Acknowledgments

We thank the German Research Council (DFG) for generously funding our research within the collaborative research center SFB 901 "On-The-Fly Computing". We also deeply thank all research assistants, all students and the administrarial staff of the SFB 901 for making OTF "fly".

References

[1] G. A. Akerlof. The market for" lemons": Quality uncertainty and the market mechanism. *The quarterly journal of economics*, pages 488–500, 1970.

[2] J. Becker, M. Kugeler, and M. Rosemann. *Process management: a guide for the design of business processes*. Springer Science & Business Media, 2013.

[3] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *Int. J. Cooperative Inf. Syst.*, 14(4):333–376, 2005.

[4] G. Burtch, Y. Hong, R. Bapna, and V. Griskevicius. Stimulating online reviews by combining financial incentives and social norms. *Management Science*, Articles in Advance, 2017.

[5] R. Casadesus-Masanell and J. E. Ricart. From strategy to business models and onto tactics. *Long range planning*, 43(2):195–215, 2010.

[6] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[7] M. R. Darby and E. Karni. Free competition and the optimal amount of fraud. *The Journal of law and economics*, 16(1):67–88, 1973.

[8] ETSI NFV ISG. Network functions virtualisation (nfv): Management and orchestration. Group Specification ETSI GS NFV-MAN 001 V1.1.1, ETSI, 2014.

[9] M. Geierhos and F. S. Bäumer. *Guesswork? Resolving Vagueness in User-Generated Software Requirements*, pages 65–107. Cambridge Scholars Publishing, 2017.

[10] J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining scalability and expressivity in the automatic composition of semantic web services. In D. Schwabe, F. Curbera, and P. Dantzig, editors, *Proceedings of the Eighth International Conference on Web Engineering, ICWE*, pages 98–107. IEEE Computer Society, 2008.

[11] M. Jakobs and H. Wehrheim. Certification for configurable program analysis. In N. Rungta and O. Tkachuk, editors, *International Symposium on Model Checking of Software, SPIN*, pages 30–39. ACM, 2014.

[12] Y. Kwark, J. Chen, and S. Raghunathan. Online product reviews: Implications for retailers and competing manufacturers. *Information Systems Research*, 25(1):93–110, 2014.

[13] J. Masche and N.-T. Le. A review of technologies for conversational systems. In N.-T. Le, T. van Do, N. T. Nguyen, and H. A. L. Thie, editors, *Advanced Computational Methods for Knowledge Engineering: Proceedings of the 5th International Conference on Computer Science, Applied Mathematics and Applications, ICCSAMA 2017*, pages 212–225, Cham, 2018. Springer International Publishing.

[14] J. A. Mills. A pragmatic view of the system architect. *Communications of the ACM*, 28(7):708–717, 1985.

[15] Machine learning library (mllib) guide. https://spark.apache.org/docs/latest/ml-guide.html.

[16] F. Mohr. *Automated Software and Service Composition - A Survey and Evaluating Review*. Springer Briefs in Computer Science. Springer, 2016.

[17] G. C. Necula. Proof-carrying code. In P. Lee, F. Henglein, and N. D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[18] P. Nelson. Information and consumer behavior. *Journal of political economy*, 78(2):311–329, 1970.

[19] G. G. Parker, M. W. Van Alstyne, and S. P. Choudary. *Platform revolution: How networked markets are transforming the economy–and how to make them work for you*. WW Norton & Company, 2016.

[20] M. C. Platenius, A. Shaker, M. Becker, E. Hüllermeier, and W. Schäfer. Imprecise matching of requirements specifications for software services using fuzzy logic. *IEEE Trans. Software Eng.*, 43(8):739–759, 2017.

[21] E. Ries. *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books, 2011.

[22] J. G. Riley. Silver signals: Twenty-five years of screening and signaling. *Journal of Economic literature*, 39(2):432–478, 2001.

[23] A.-W. Scheer and M. Nüttgens. Aris architecture and reference models for business process management. *Business Process Management*, pages 301–304, 2000.

[24] I. V. Serban, A. Sordoni, Y. Bengio, A. C. Courville, and J. Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. In D. Schuurmans and M. P. Wellman, editors, *Thirtieth Conference on Artificial Intelligence (AAAI)*, pages 3776–3784. AAAI Press, 2016.

[25] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[26] C. Stummer, D. Kundisch, and R. Decker. Platform launch strategies. *Business & Information Systems Engineering*, Articles in Advance, 2017.

[27] D. J. Teece. Business models, business strategy and innovation. *Long range planning*, 43(2):172–194, 2010.

[28] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In I. S. Dhillon, Y. Koren, R. Ghani, T. E. Senator, P. Bradley, R. Parekh, J. He, R. L. Grossman, and R. Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pages 847–855. ACM, 2013.

[29] J. A. Zachman. A framework for information systems architecture. *IBM systems journal*, 26(3):276–292, 1987.

[30] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.

[31] S. Zimmermann, P. Herrmann, D. Kundisch, and B. R. Nault. Decomposing the variance of consumer ratings and the impact on price and demand. *Information Systems Research*, Articles in Advance, 2017.