IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 20, 2020, accepted August 22, 2020, date of publication August 28, 2020, date of current version September 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3019985

On Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices

TOMAS CERNY[1], JAN SVACINA[1], DIPTA DAS[1], VINCENT BUSHONG[1], MIROSLAV BURES[2], PAVEL TISNOVSKY[3], KAREL FRAJTAK[2], DONGWAN SHIN[4], AND JUN HUANG[5], (Senior Member, IEEE)

[1]Department of Computer Science, Baylor University, Waco, TX 76798, USA
[2]Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University, 121 35 Prague, Czech Republic
[3]Red Hat Czech, 612 00 Brno, Czech Republic
[4]Department of Computer Science, New Mexico Tech, Socorro, NM 87801, USA
[5]School of Computer Science, Chongqing University of Posts and Telecommunications, Chongqing 400065, China

Corresponding author: Tomas Cerny (tomas_cerny@baylor.edu)

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research https://research.redhat.com.

ABSTRACT Code analysis brings excellent benefits to software development, maintenance, and quality assurance. Various tools can uncover code defects or even software bugs in a range of seconds. For many projects and developers, the code analysis tools became essential in their daily routines. However, how can code analysis help in an enterprise environment? Enterprise software solutions grow in scale and complexity. These solutions no longer involve only plain objects and basic language constructs but operate with various components and mechanisms simplifying the development of such systems. Enterprise software vendors have adopted various development and design standards; however, there is a gap between what constructs the enterprise frameworks use and what current code analysis tools recognize. This manuscript aims to challenge the mainstream research directions of code analysis and motivate for a transition towards code analysis of enterprise systems with interesting problems and opportunities. In particular, this manuscript addresses selected enterprise problems apparent for monolithic and distributed enterprise solutions. It also considers challenges related to the recent architectural push towards a microservice architecture. Along with open-source proof-of-concept prototypes to some of the challenges, this manuscript elaborates on code analysis directions and their categorization. Furthermore, it suggests one possible perspective of the problem area using aspect-oriented programming.

INDEX TERMS Code analysis, distributed systems, enterprise architecture, enterprise systems, global governance, microservice,

I. INTRODUCTION

Code analysis has an extensive history and scope of research [1]. It can be recognized for benefits, including automation in areas such as code clone detection, error detection, malpractice detection, formal verification, security evaluation, quality assurance, reverse engineering, etc. However, in our opinion, the development practices for large enterprise systems are underrepresented in code analysis research and deserve special attention. Over the past few decades, many design best practices and standards have been established and applied across different enterprise platforms. However, the mainstream code analysis research focuses on low-level program aspects that, while essential, cannot assist to many *enterprise* challenges introduced by the growing complexity, standards mentioned above, and practices applied in software design along with recent design demands for distribution and cloud computing.

What is commonly seen as an enterprise [2], [3] are large and complex systems interacting with many users, enforcing various business constraints or processes. Enterprise software is thus an application that a business would use to assist the organization in solving enterprise problems. Enterprise solutions bring automation into many disciplines, including healthcare, transportation, telecommunication, banking, e-commerce, power grids, and defense systems, among others. Without a doubt, modern enterprise software solutions are drivers for our present and future economy. When we

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaobing Sun.

apply ''code analysis'' instruments into enterprise software, we receive reports with too low-level information. This is because current instruments do not aim to understand common development standards and the direction of enterprise software design. Let us ask the following question. If we were to build a large enterprise solution, would we use common standard edition language to develop it from scratch, or would we take advantage of enterprise development frameworks? Most likely, we aim to avoid wheel reinvention, and our budget is limited; thus, we build on top of existing best practices and standards to reuse preexisting components and involve appropriate enterprise architecture [2]. In Section II, we will look into the background and what a modern enterprise architecture looks like and what is the target deployment environment these days.

This manuscript aims to provide a road map to selected motivating challenges in enterprise software, which could be well addressed by code analysis that recognizes enterprise constructs, components, and standards. This work looks at both monolithic and distributed solutions, covered in Sections III and V. The typical analysis techniques are discussed in Section IV. Besides, this manuscript provides a possible problem formalizing perspective through distributed Aspect-Oriented Programming (AOP), which fits well to describe the problem distributed enterprise application design. This perspective is detailed in Section VI. This manuscript lists our preliminary open-source instruments to multiple of the mentioned challenges offering a proof of concept. In the Appendix, we also provide links to the open-source repositories to simplify the research initiation. After reading this manuscript, one will gain a solid overview of current enterprise challenges well suitable to be addressed by code analysis. Moreover, the reader should better understand how code analysis can fit the mainstream industry directions and why one should pay attention to it.

In grasps, one can expect to learn how to face information restatement across application layers [2] or redefinition of concerns when code analysis is given more responsibility to generate or derive code fragments. Next, it provides a path to identify inconsistencies and contradictions in security policies or data constraints. Besides the challenges apparent in monolith systems, elaborated in Section III, the core focus of this work is on distributed enterprise solutions. In particular, we consider the recent mainstream microservice architecture for distributed enterprise solutions. As one will learn in Section V, this architecture brings interesting challenges. Since we found it useful in our previous research to apply AOP for the distributed solution challenges, we dedicated Section VI to it. The further possible directions implied from this work are to be described in the concluding section.

II. WHAT IS AN ENTERPRISE ARCHITECTURE

Before we describe enterprise architecture, lets first discuss what software architecture is and what perspectives we might want to differentiate. Bass *et al.* [4] suggest that ''software architecture of a program is the structure of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them,'' however, alternative definitions exist [5]. In their book, they see system functionality orthogonal to other system properties, and thus the same functionality can be implemented involving different software architectures, implying different system quality attributes. There are many architectures that the system can utilize so that enterprise architecture can be seen as a subset of the possible architectural options. A subset of architectures that fits the most common enterprise use cases (e.g., data management).

If we consider another sort of architecture, a building assembly scheme, electrician experts will most likely look for different information and detail in the plan than the plumbing export, kitchen specialist, or the general builder. Similarly, one may like to have a similar architectural view in software engineering. Larman [6] suggests using an N+1 view model to document various architectural perspectives, such as logical view, processes, data flow, security, etc. Thus, software architecture might be of interest to different person roles aiming for different goals.

A significant contribution to the enterprise domain, along with the best practice, is highlighted by Fowler [2]. In his book [2] he gives a sense of what is and what is not an enterprise architecture providing examples: ''Enterprise applications include payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading. Enterprise applications do not include automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.''

One could hardly imagine enterprise applications without processing persistent data kept for a long time or indefinitely. Such data are typically processed by one or more involved parties and may change over time. One could see such systems as data-centric. Data and its specific subsets might be accessed from different parts of the system by distinct user roles. Data typically remain even when the system software upgrades. We can typically expect enterprise applications to involve databases to manage the data. The integrity rules, business logic, and knowledge of what to do or how to process data are encoded in the enterprise application.

We can expect that enterprise applications deal with a lot of data concurrently. Besides, they provide user interfaces or at least endpoints to machine process the data (e.g., middleware). We can also expect enterprise applications to integrate other third-party enterprise application modules, leading to system integration. Wide heterogeneities might be expected across particularly involved modules when dealing with system integration. Perhaps the most critical part of each enterprise application or a module is its internal knowledge expressed by data scheme and business logic and rules specifying what is allowed and what is not and what actions are triggered if some event happens. Fowler [2] ironically mentions that large systems eventually grow into complex

business "illogic" since different people define many rules and perhaps distributed (non-centralized), and follow different implementation approaches. The single basic rule change may lead to unexpected consequences that demand thorough testing, evaluation, or verification.

A. ENTERPRISE DEVELOPMENT FRAMEWORKS

Fowler [2] also mapped the existing enterprise application framework architectures into three layers. These layers have a specific focus and are common across frameworks until today. We will use Java Enterprise Edition (EE) terminology [7] to describe layers, namely persistence, business, and presentation layers. The persistence layer typically addressed the technological incompatibility between object-oriented (or component-based) design and data storage, often a relational database. This layer defines data schemes and deals with core performance services, message systems, and transactions. The business layer defines services on top of the data schemes. It contains the business logic and constraints along with security authorization. If the system also provides frontend, then there is data instance mapping in the presentation layer into presentation frameworks [8], controllers restricting access control, and management of the presentation logic. Modern enterprises often separate presentation into client-based frameworks, such as Angular [9] or React [10], and the presentation layer reduces to endpoints for the client-based user interface running and managing the state in web browsers. It is not intended to describe every edge case of an enterprise in this background section, but rather to provide an idea of what is the mainstream direction in design for enterprise systems. Clearly, the three-layers architecture greatly describes what an enterprise application may look like with recent trends. It applies to particular modules of the overall system.

Enterprise development frameworks usually provide components to speed-up development. Such components might be extended classes used for specific purposes, typically accompanied by meaning descriptors. For instance, a persistence component involving annotations or XML descriptor indicating that given class mirrors to a database involving object-relational mapping. It is important to note that plain classes would be very low-level for enterprise design. Frameworks bring specialized components and constructs, e.g., to deal with data persistence, REST services, messaging, transaction management, security enforcements, input validation, etc. [11]–[19]. This is more or less the case for enterprise frameworks for various programming languages. While some of these enterprise standard references might seem old when we consider their original release date, it must be noted that they are still in common use as best practice. For example, the most recent Java enterprise platform release [7], [11] uses them, and trends show that these standards will remain current for the foreseeable future. For other platforms, consider the similar instruments and mechanisms, such as Object-Relational Mapping (ORM) in the .Net platform using an entity framework [17]; Python with frameworks SQLAlchemy, Django, or Tortoise [18] or PHP Doctrine [19] and Lavarel [20], and Java Persistence API (JPA) in Java [21]. If we wanted to understand the module internals, understanding the involved component and their constructs is essential. This is the driving force for this article, where we aim to motivate the researchers to consider the recognition of these constructs and their intent for code analysis. For instance, with ORM constructs, we can easily recognize which data the system operates with, what is the data identifier, required fields, etc. With plain code analysis, this might be more challenging.

B. THE RISE OF MICROSERVICE ARCHITECTURE

All that was described so far in the previous subsection is quite a standard for almost two decades, and the recent enterprise direction pushes towards mass scale, cloud computing, and distribution. A few years ago, the directions for distributed system integration were to modularize functionality to services and use Service-Oriented Architecture (SOA) [22] involving centralized Enterprise Service Bus (ESB) [23] to route messages and connect services. The processes and constraints could be centralized through the ESB. In case a new service emerged, it could have been easily integrated into the existing process flow. However, while initially intended for distribution, this architecture led to monolithic deployment, draining many efforts and often using single schema with canonical data model perspective and centralized management. This did not give developers much freedom to evolve particular modules independently [23]. In literature, it would be hard to find supporting evidence that SOA was cloud-friendly [24], which is easy to state for what we mention in the next paragraph.

For the above drawbacks and several other reasons industry pushed for novel distributed and cloud-friendly Microservice Architecture (MSA) [3], [23]. MSA is driven, in large part, by the demands of scalability and heterogeneous data integration. MSA based on three principles [25]: "A program should fulfill only one task and do it well; programs should be able to work together, and programs should use a universal interface." The core difference from SOA is that microservice architecture is a share-as-little-as-possible architecture pattern that places a heavy emphasis on the concept of a bounded context. In contrast, SOA is a share-as-much-as-possible architecture pattern that places heavy emphasis on abstraction and business functionality reuse [23].

The overall MSA system is divided into multiple heterogeneous, self-contained, self-deployable modules interacting over remote calls or through messaging. This structure provides many benefits. Multiple teams can work on individual modules, and they can adopt different languages and frameworks. Since modules are self-contained, they can be dependent on different library versions. Still, mostly, they can be deployed, managed, and evolved individually. Modules are designed to have limited coupling amongst each other, which limits global coordination activities, which is easy to do in

monolith-like systems. This would have been necessary for the past using the SOA involving ESB [23], [26].

C. PRODUCTION DEPLOYMENT OF MICROSERVICES

To complicate matters further, big industry players have various supportive deployment platforms [27]–[30] for cloud-based container orchestration. These provide scaling functionality, routing, monitoring, etc. Typically, these platforms provide MSA module isolation within pods, wrapping each module in a container [27], [30], [31]. Each pod is further wrapped in a node that includes a load balancer for external requests. These constructs are not merely overhead; they directly support scalability. One pod can be easily replicated into identical pods to adjust to demands elastically, leveraging cloud computing benefits.

According to the NGINX survey [32], most businesses are building on top of the MSA, which has become entrenched as a flexible model for a broad set of industry problems. Subsequently, the market for microservices continues its rapid growth; by 2023 it could reach $33 billion [33] with an annual growth rate of 16-17% [34]. Microservice architecture can take full benefit of cloud computing, enabling rapid scalability and performance. Typically, a mesh of hundreds of services [35] can exist, providing individual service monitoring and recovery, being perfectly elastic towards request demands, and thus optimizing hardware utilization and performance. In this architecture, individual microservices are self-contained modules, isolated from one another to enable independent deployment. The question is no longer how to scale such systems, but how to assess them from various perspectives to understand the global picture of the overall system.

Based on online resources [36], [37], at the end of 2019, Kubernetes was the first choice among developers for a container environment, with 45% usage. According to another large survey [38], Kubernetes was used in production by 78% of respondents. It has over 60 thousand stars at GitHub. Tinder uses Kubernetes to run 48,000 containers. Among other users are Reddit's, The New York Times, Airbnb, and Pinterest (with 250 million monthly active users), Pokemon Go (with 20+ million daily active users), and many others. The typical organization runs about 14 containers per host with Kubernetes. In the Azure cloud, approximately 65% of organizations using containers also ran Kubernetes in 2018. According to [39], over 80% of companies using containers also use container orchestration or a service mesh.

A service mesh offers automated load balancing, service monitoring with automatic metrics and log and trace management for traffic, discovery, security, and failure handling. To add these supportive features, each MSA module has a companion sidecar proxy intercepting traffic. Example meshes are Knative [40] and Istio [29], which both build on top of Kubernetes. Service meshes or cloud-based frameworks are typically monitored. Prometheus [41] is an example, an open-source monitor. There exist many metrics exporters for Prometheus from third-party systems that are also open-source, which simplifies extension and adaptation. Apart from this, Grafana [42] is an example of open-source analytics and monitoring solution that can query matrices from Prometheus and visualize them. The 2019 survey mentioned above [38] suggested that 72% of respondents used Prometheus in production.

D. GENERAL CHALLENGES WITH MICROSERVICES

We cannot expect the industry to change the current mainstream direction and practice. However, we can expect growth in complementary or supportive mechanisms. For instance, excellent complements are serverless functions that provide finer decomposition granularity than MSA modules. Serverless functions are an excellent fit for real-time reactive systems when events are processed on a large scale. Consider a use case for per-user request handling where a user posts a review, including a video that requires formatting. As opposed to MSA, serverless functions accommodate small module features that need to scale separately.

One can expect the growth of new tools to address some of the significant challenges that arise with this architecture, stemming from isolated module design and module heterogeneity. It is important to note that there are two dimensions of module granularity. The overall system is divided onto multiple loose-coupled heterogeneous modules, e.g., one dealing with user accounts, another dealing with payments, and a third with orders. Next, there are module replicas where identical modules co-exist to provide better scalability of module-specific features.

The module coupling is loose but still exists across the heterogeneous modules since they interact with similar data. Modules see certain data within their specific bounded context. For instance, multiple modules may operate with a set of users; however, one module may need a user's address, another might require the user's credit card detail, and another the user's email. Since modules evolve individually, there is a constant emergence of maintenance challenges. A change in one module can impact other modules with respect to data structures (e.g., new fields), business constraints (e.g., field length), or even security enforcement (e.g., restrict method access). Modules developed by individual teams become heterogeneous over time as modules continually evolve, and it becomes harder to build the global picture.

Software architects who aim to optimize the system as a whole need to understand each module's internal characteristics and thus be able to locate important module aspects in the code, but given the possible mix of languages or frameworks, it becomes hard to assess the overall system by hand and perform global optimization and consistency verification. Even routine testing becomes a burden [43], [44].

In today's enterprise environment, the same developer would rarely develop, deploy, and upgrade all the modules in a system. Each developer or development team works on its assigned module, deploying it locally to debug and test. The production environment is managed by System Operations (SysOps) personnel who have extensive knowledge about

the service mesh and deployment process. The issue is that SysOps typically are given each module with limited information about its specific needs, and thus they would benefit from easy access to the module's salient aspects to understand deployment specifics. Similarly, developers have a limited understanding of the deployment infrastructure, which can result in system dysfunction.

To recognize architectural issues and inconsistencies, we would first need to understand both the system and the module's source code, so that particularly involved structures, data, and control flow can be identified. Such an assessment would demand much time when doing it manually. Second, to make suggestions for the runtime performance, we would need sufficient monitoring, providing thoughtful details for particular, observed parts of the system or access to logs to reconstitute the interaction details from involved modules. Finally, we would need to have access to system resources and use statistics, e.g., CPU utilization at a given time, usage of memory, or network usage for runtime performance optimization. However, performing all these analyses by hand is cumbersome and time-consuming.

This begs the driving question: How can be the process of understanding a system or individual modules sped-up? We argue that this is possible through novel code analysis. Traditional code analysis [45]–[59] looks at the low-level program language constructs and thus overelooks important enterprise development framework constructs. It fails to identify important system components. Only few approaches consider the enterprise details, including our previous work [60]–[65] which we detail in later Section.

III. CODE ANALYSIS IN MONOLITHIC ENTERPRISES

A. THE MOTIVATION - LESSON FROM THE ORM

The most notable challenge in enterprise systems in the early days was ORM. Developers had to deal with mapping of data entities represented as classes to SQL tables in relational databases. Any time data were retrieved, they were mapped to objects and back upon persistence. Fowler lists dozen of patterns for ORM [2]. From today's perspective, there is a standard for ORM for Java EE platform [11] JPA [21] recognized more or less by other programming languages as well. ORM frameworks simplify the mapping so that one does not even need to manage SQL tables anymore (exaggeration, since optimization and custom indexing are often needed).

One additional benefit of ORM is that its metadata can be used for verification purposes. For instance, towards the database schema. Upon application initialization, the class setup can be checked towards the tables, columns, or constraints in the database schema, and thus any inconsistency can be detected. If there is an inconsistency found, the application initialization fails. Alternatively, we let the ORM perform an update of the schema to match the class model.

In order to recognize the class settings, JPA audits each class involving the class meta-model [66]. In the case of Java and C#, it is easy to involve introspection since both languages support reflections [67], [68]. In PHP frameworks similar mechanism is to use comments [19], alternatively to use XML class descriptors. In particular, it considers class name or its annotated properties, each field along with its data type and properties, and matches particular associations or even inheritance with appropriate mapping strategy [2], [69]. ORM solves the technical inconsistency between Object-Oriented Paradigm and Relational Databases, and nowadays, framework solutions integrate it with many best-practice patterns [2], [70] improving the caching, performance, type safety and efficiency. With ORM, one can select a particular relational database and handle the majority of use cases; for fine-tuning, one may need to introduce custom optimized queries [71], but from our practice, this is rather marginal.

Since ORM solves very well the consistency issues across different paradigms, could it be used elsewhere? Indeed, however, first, it is essential to note that ORM is not platform-specific. As long as we involve Object-Oriented design, ORM is applicable. This can be easily demonstrated by the ability to map JPA onto UML class diagrams [66], [69]. To do that, one can involve UML stereotypes and introduce the UML profile to extend class diagrams. Having ORM described on the UML level is just a matter of implementation to transform it into a particular domain. Since the object-oriented design is the mainstream, it will fit most modern frameworks.

B. USER INTERFACE DERIVATION

Where else could be applied the approach similar to ORM to bring consistency benefits and a single focal point? One challenge with enterprise systems actual till these days is the overhead, coupling, and restatements between application backend or middleware and the User Interface (UI) part. UI development in enterprise systems consumes around 50% of the overall project development time [61], [72]. If we thought that ORM is a great solution, we might now think that it helps only with a small portion of what the development of the overall application involves. The percentage is even more significant when we talk about adaptive UIs. Besides the high development efforts for the UI part, there is also a secondary issue. The issue is the coupling and restatement between the UI code and lower application layers, mostly the data scheme.

For instance, if we have a person form binding to person objects, at any time, we change the person's class, the form must change as well. Next, if we want to show the form in a single- or two-column layouts, highly likely, we might need to develop too separate forms. If we had to support the mobile version of the system and web clients, we grow the replication of forms using distinct native components.

However, there is an additional problem dimension - a restatement. If a person has a restriction that the email address field must be an actual email format, the user interface typically lacks a mechanism to reuse this constraint already defined at the backend and restate it again in the UI code. If the backend changes the restriction, there is no mechanism

to detect that the UI is inconsistent with the backend. These UI challenges are quite similar to the ORM problems.

In [60], [61], [63], [66], [72]–[74] authors suggest to derive UI fragments related to data presentations automatically. In particular, it uses existing information captured by the class data model and involves ORM annotations [75]. Besides ORM annotations, there is also an enterprise standard for input validation that is also captured on data classes [13]. Also, recently was released security restricting role-based access control [16]. In this approach to derive UI, the particular frameworks perform in three stages: inspection, field template selection, and template population. The initial step to inspect data classes uses reflection API mechanisms for introspection. Next, based on each field property, it selects a particular template, utilizing configured transformation rules. A UI field template comprises the UI code involving the target language (e.g., HTML, Angular, React, JSF, etc.) and markup expressions/variables meant to populate the data-specific details. The template content is resolved as follows. The template is parsed, and the expressions are found and resolved. In particular, the expressions typically reference the properties of currently processed data class field properties, such as its structure, ORM settings, or input validation, which is part of the processed data class/field.

For instance, to derive one form line for a textual data class field, a template is selected with standard text input. If the anticipated length is below 255 characters, it selects standard text input. Otherwise, a text area component template would be selected. Next, if there are any constraints, such as not empty property or required format for the content (e.g., email), the validation is added to the template based on an expression in the template (e.g., onBlur JavaScript listener). This sort of transformation is applied to all the filed of a given data class. It guarantees always to produce a form matching the data class. This process avoids consistency errors between the backend and frontend of the application. Once the data class changes, the derived form will reflect the change. This would not necessarily be the case when done manually. Unfortunately, a manual approach is still a typical case in conventional development practice.

C. CONTEXT-AWARE USER INTERFACE DERIVATION

One could argue that derived UI data presentations are too monotonous, and this could seem to be a limitation or a reason why it is not used on a large scale by developers. For instance, what if we wanted to develop a context-specific UI? Such adaptive or personalized UI fits specific user needs or conditions rather than providing the one-fit-for-all, mostly a partial fit for all users. In a manual approach, this would lead to the development of many separate UI fragments of the same data presentations opening enormous demands for maintenance and evolution. However, how would the generative approach fit here? The answer to this question is elaborated in [62], [63]. It is no longer the case that only reflection is necessary. To enable adaptive UI, the derivation process must consider a more generic solution that allows us to effectively deal with multi-dimensional problems, ideally each in a separate dimension. AOP well addresses this since it is a cross-cutting concern. This also means that the derivation process must extend accordingly.

AOP divides the multi-dimensional problem into multiple separate dimensions, which are later combined by aspect weaver. Typically, programs comprise the main/core logic, which can be further extended as we add the additional dimensions through aspects. Without AOP, the developer is the one to do this weaving in the source code, which then results with a spaghetti-like code that tangles individual problem dimensions and makes each dimension restated and non-reusable. Basic object-oriented programming focuses on the design of an individual situation rather than a generic multi-dimensional problem. Thus, AOP recognizes objects-design for the core logic and aspect construct for the additional dimensions that may extend the core logic. This makes the addition dimensions reusable and untangles the code. One specific situation is then not a simple code interpretation as in object-oriented code but multi-dimensional assembly involving the core object-oriented code part and separately defined aspects. This way, each situation is derived based on what the individual dimension specifies. If one dimension changes, all the resulting code is influenced, which nicely enforces those changes to all the system products/parts rather than asking a programmer to changes the particular dimension perspective in each developed custom situation that restated the dimension definition.

In [62], [63] a generic aspect-weaver for UI is introduced. It differs from the straightforward UI derivation in the flexibility towards customization and the transformation process stages. In the first step, information from the backend is collected through code analysis. The overall code is too broad and detailed; thus, a simplified representation of the code is derived. In AOP [76], this representation is known as the join point model. It comprises the structure of recognized data elements and all their details, including fields, data types, and annotations. These all can be seen a join points. However, it is important to note that these are static join points, and this will not change in application runtime.

Along with these, dynamic join points can be considered, for instance, the origin of the particular user, time, data settings, screen size, etc. Dynamic join points are time-specific and recognized at runtime. The stage of join point model collection thus partially happens once for the static join points and at runtime upon request for the dynamic join points. The typical code transformation would move forward to populate templates statically. However, the AOP approach delays the assembly until runtime. It also inserts an additional stage that determines the UI component templates based on the join point model and grammar-based expression rules. The idea here is that developer develops these mapping rules based on the join points, which act as variables in the expression language. In [62], [63], Java Unified Expression Language (JUEL) is used to take advantage of existing well-recognized language interpret that determines

whether a particular expression is met (resolved to true) and if so a particular UI component template is selected. This is the code engine for the aspect-weaver. A particular template that is selected based on the expression rules can be seen as an AOP *advice* for the *pointcut* given by the particular expression rule when using the AOP terminology [77]. Such resolution happens for all particular data fields identified at the particular data entity. The next stage is similar to conventional UI derivation to resolve the template content. However, it can base on runtime join points along to the static ones that are accessible as template variables. Similarly, to the definitions of template selection rules, the developer is in charge of making component templates; however, it is not an exhaustive job. An example enterprise production system with hundred of data entities, in [63], needs only seventeen templates that are generally applicable. This can be seen as an alternative to restated component settings for almost eight hundred fields. It shows clear benefits through significant code volume reduction, improving development speed, maintenance, and consistency with marginal efforts. The process may also incorporate additional stage weaving the resolved field templates into a layout template. Again, the layout might be generic or data entity-specific identifying field positions or order. A particular layout selection is resolved based on join points (e.g., based on detected screen size, count of fields, etc.).

D. ADVANCED OPTIONS IN UI DERIVATION

Extended benefits sourcing from the AOP process are discussed in [62], [64], [78]. When we consider modern but conventional approaches when one middleware serves multiple different frontends, e.g., web-based, desktops, or mobile, most likely each managed by a different team, the consistency issues emerge with the middleware evolution. This is a serious problem since changes to the middleware done by one team might not be articulated to other teams responsible for the frontend, which leads to consistency errors and possible security issues. With the AOP process to derive UI's, we can separate and distribute the weaving process onto platform-independent and platform-specific parts. This brings significant benefits and addresses the above-stated problem. In particular, the middleware can be assessed by the middleware aspect-weaver, derive the join point representation and provide it in platform-independent format to the client weavers. Client weavers feed on the platform-independent joint point representation. They are platform-specific (web-based, desktops, mobile, using a given framework or platform) and specify the UI component templates for a given platform along with templates for layout. When changes are made to the middleware, the middleware aspect-weaver observes the changes which ensure that all client weavers reassemble the UI fragments to preserve perfect consistency.

An energy-related observation is shared in [78]. In client-server interaction, we typically send the UI in the form of HTML or JavaScript and CSS. However, it must be noted that typically when the server provides the User Interface to clients, it tangles different information concerns together (e.g., layout, fields, constraints). This increases the replication in resulting HTML leading to larger transmission volumes (possibly amortized by transfer compression). When we consider the distributed aspect weaving with middleware and client-based weavers, then we observe that particular concerns remain untangled throughout the separate delivery. This facilitates parallelism and also enables us to apply different caching strategies to particular concerns. This can have even greater granularity and benefits than client-based technologies such as Angular or React. Any energy impact assessed in [78], showed that the aspect-based UI approach is a more energy-efficient approach when compared to conventional server-side or client-side UI technologies.

What if another significant system concern could be added? Such an interesting perspective in enterprise systems is business rule management. Unfortunately, there is no unified approach to capture business rules. However, it is common to capture these rules in the business layer in service objects. There are interesting rule engines enabling developers to use domain-specific languages, such as Drools framework [79]. If one manages to capture business rules in such a unified approach, it is possible to extract business rules and consider them for transformation or even the AOP process for UI fragment derivation. For instance, [80], [81] shows that it is easy to extract business rules from Drools and convert them to documentation or even enforce client-side validation already in the UI. This could be utilized for knowledge base exchange across smart interacting systems.

E. SECURITY POLICY ASSESSMENTS

Another interesting issue in enterprise monolith applications where code analysis can significantly help is security policy assessment. For instance, Role-Based Access Control (RBAC) [82] is commonly used, and even standardization exists [16], [83] for enterprise solutions in the form of annotated endpoint methods restricting selected user roles to perform calls. So far, we mentioned a reflection mechanism that can identify existing structures, annotations, and properties. However, to perform security assessments and identify inconsistencies across system resources or call paths, the reflection mechanism is insufficient. There are other two greater approaches, source code analysis and bytecode code analysis (or binary code analysis). The first approach requires access to the application source code and uses language parsers commonly available for various languages. The second approach does not require access to source code and utilizes libraries for reverse engineering or bytecode code manipulation; we briefly introduce the code analysis strategies in the next section.

Regards security assessment, both source code analysis, and bytecode code analysis is equivalent when one wants to extract control-flow or to identify important system resources involved in a given control-flow. Typically, we first identify all system endpoints. This task is rather easy in enterprise systems, and we detail the process in the next section. For

each endpoint can be extracted control-flow graph involving method calls. Using the identified graphs, one may identify the graph overlay. Since each path is associated with a particular user role given by the endpoint, we detect inconsistent paths that lead to the identification of endpoints that apply inconsistent security measures to reach certain parts or resources of a given system. This approach was evaluated in [84], and lead to the identification of five common security issues with, e.g., inconsistent roles from different endpoints, role hierarchy violations, or even inconsistent access to data entities as an example resource access. Involving code analysis extended to recognize enterprise constructs, we can verify security consistency, and detect security vulnerability within the system.

F. OTHER CHALLENGES

If we were not considering enterprise systems, the number one research topic for code analysis would be code-clone detection and software error detection. There are numerous approaches involving source code analysis [59], [85], [86] or byte code analysis [87]–[89]. However, these approaches use low language constructs only. If we looked for approaches recognizing enterprise components, we would not find many publications. However, the needs here are great. For instance, the enterprise components provide more context to identify the semantics, as suggested by [90]. This could lead to better identification of semantics clones, which are usually challenging to detect in low-level code analysis [91], [92]. These semantic clones are harmful [85], [93] since they lead to consistency errors during system evolution. Existing works on semantic clones [94]–[104] are approximations as it is a formally undecidable problem [59], [105] and thus additional semantics brought by the enterprise components and construct can greatly simply the detection, bring new insights and speed-up the process.

One significant fact was left unaddressed in this section. Specifically, the fact that most modern enterprise systems are distributed. An entire Section V is dedicated to distributed enterprise systems. However, before we move to it, we introduce more details on the different code analysis techniques mentioned briefly in this section.

IV. CATEGORIZATION FOR ANALYSIS TECHNIQUES

To analyze software systems, we might be interested in techniques and approaches that have been considered previously. However, it might also be useful to consider some structuring in these techniques. We provide a basic categorization considering different kinds of inputs, language types, and analysis realizations. We also categorize what is typically analyzed in software systems.

Code analysis typically enables us to recognize particular constructs such as components, classes, methods, fields, or annotations and to utilize relationships among them. There are many different types of components in enterprise platforms, such as controllers, entities, or services. Typically we can involve language parsers that tokenize the code and represent it in a graph format. However, there might be specific information we aim to collect in the analysis.

A. THE NATURE OF THE INPUT

The first perspective we consider in analysis categorization is the *nature of the input*. A software application may consist of application code or configuration files, and both can be subject to the analysis. For instance, it is common for modern applications to consider deployment descriptors or build files that both can contain important information, such as specification or remote endpoints, and deployment details related to the containerization [30] and its dependencies. When we look at the code, which is the most obvious direction, there might be a general application code but also code fragments that use a specific syntax common for a particular task in the overall system.

B. LANGUAGE DOMAIN SPECIFICITY

To elaborate on this perspective, we can *categorize languages* onto General Purpose Languages (GPL), such as Java, C#, or Python and Domain-Specific Languages (DSL) [106]. The nature of GPL is a general-purpose, which means we can do everything but perhaps not most effective for every task. Assuming that GPL nature and purpose is obvious, we dedicate more detail to the DSL description. DSL are languages specific to a particular domain or task. For instance, in enterprises we can use DSL to describe processes [106] or rules [79]. It can be business rules [80] or security rules. Modern frameworks also make it possible to define specific processes using DSL and utilize modeling tools on top of the DSL [107]. For instance, Java EE [11] specifies Batch processing that involves DSL, and integrated development environments provide models on top of it. Besides, we can use DSL to define reports through robust templates quickly or use DSL for the deployment descriptors aiming to simplify the description process. We can even define our special-purpose language to simplify specific tasks [108]. DSL is typically more efficient than GPL for the specific task they target. However, they have to be interpreted or compiled separately from the main code. Many DSL bases on XML which makes the processing simple as many parsers for XML exist. DSL has its fit within AOP. In particular, it can be a great fit to describe aspects or advices [63].

C. LANGUAGE STAGE USED FOR THE ANALYSIS

Another perspective we consider is in which language state is the analysis considered and realized. We consider in which language stage do we perform the analysis. All languages have source code that can be subject to the analysis. However, some languages compile into an output that can be subject to the analysis. Moreover, some languages contain a mechanism to describe themselves called introspection and involve metamodel. There are certain benefits for each of the particular strategies, and there are use cases when one may fit better than others.

1) INTROSPECTION

Metaprogramming [68] gives a program or a language the ability to examine or modify both its structure and behavior at runtime. It can effectively extract information related particular components. In particular, it is capable of observing object properties, methods, annotations, or other particles. Many contemporary, statically-typed programming languages can describe them, called Reflection [67]. Besides, reflection is even capable of modifying the behavior. A given object can be examined by only a subset of the reflection abilities. This is called introspection [45]–[48], [63], [68].

With introspection, we could effectively extract information related to software data schema, endpoints, authorization restrictions or input validation [60]–[63], [72], [78], [81], or even particular types of components given by annotations. The most significant limitation is that introspection, as offered through reflections, does not provide all program details. It does not supply details about the method body, which carries vital information to derive control-flow or dependencies. We can find reflections in Java, C#, PHP, Python, etc. However, in C++, a third party library would be necessary. For instance, with a preprocessor generating descriptors for classes, fields, and methods. Another disadvantage is that any system extraction logic must be part of the system. Though, this could be solved through the integration of a generic library [63].

2) BYTECODE ANALYSIS

Another possibility is to analyze code which is in its intermediate representation [45], [47], [48], [50], [60], [61], [63], [72], [109]. Multiple programming languages compile the source code into intermediate representation to support various platforms, e.g., bytecode. Java, Go, and Python languages allow performing bytecode analysis [110], [111]. Researchers have used this approach to generate test scripts [47], [109], generate testing data [50], extract models to better understand legacy systems [48], to perform formal verification of low-level programs [46], [51], or to implement defect detection tools [112]. Bytecode analysis is often used to check errors or possibly also to detect code-clones [52], [53].

Bytecode analysis [51] needs the compiled code of an application. Like a reflection, it can uncover components endpoints, authorization policy enforcements, classes, methods, and another context including data identification, control-flow, method calls, or even Abstract Syntax Trees (AST) [52], [53], [113]. Bytecode analysis works well for languages such as Java or Python. However, the disadvantage is that not all languages use bytecode.

3) SOURCE CODE ANALYSIS

Source code analysis [54] takes the source code of the system, parses it, and builds its graph representation. Typically, there exist parsers for existing programming languages. Parsed programs are represented as a parse tree or an AST [85], [114]. The advantage of this category is that we can use it at any time. Besides, this is the approach to consider for DSL along or for the build and deployment descriptors.

Source code is commonly parsed and transformed into graph representation to perform pattern recognition [49], [50], [55], [57], [115], [116]. It is common to use source code as the input for quality assurance tools [55], [116]. Besides it is a common analysis used for code clone detection [49], [54], [117]. Source code analysis can access full details similar to bytecode analysis. However, there exist situations when we do not have access to the source code and only posses the application bytecode (e.g., application deployed on a production server).

The source code analysis can also be utilized for Mining Software Repository (MSR). This involves accessing the source code from the version control [115], [118]–[120] and considers elements such as classes, functions/methods, data structures, return values, the control-flow, particular statements, code comments, or version control metadata and statistics.

D. COMMON ANALYSIS GOALS AND STRATEGIES

Once we identify the right language stage for the analysis, we continue to identify our concern. Here, we categorize goals and strategies observed and assessed in existing works. First, we look into graph representations. Next, we consider the identification of individual components, flows, and patterns. Certain tasks involve the localization of specific artifacts in the system. Many analysis techniques consider metrics, and apart from this, some strategies consider transformations. We want to underline that these categorizations are not exclusive and are meant to provide a route map to the analysis techniques.

1) PROGRAM GRAPH REPRESENTATIONS

When we parse source or bytecode, we tokenize it and derive AST. However, alternative graph representations can be used. For instance, to represent method calls, what has been adopted is a call-graph typically a Control-Flow Graphs (CFG) [50], [121], [122]. However, graphs are also utilized to draw dependencies across artifacts in the code. Program Dependency Graphs (PDG) [85], [114] has been used for such dependencies.

For introspection, we would still derive trees representing particular perspectives. For instance, in [63], we used a tree to represent each data entity with its attributes and descriptors. In our research practice in the enterprise domain, we used both bytecode analysis and source code analysis to recognize AST, CFG, and PDG. Typically these graphs would be extended with API method details, various component types, annotations, etc. We experimented with Java and Python and considered enterprise framework constructs along with enterprise standards [11]–[16], [18] to prove feasibility of utilization of enterprise constructs in code analysis. We like to highlight that there is no approach superior to others, each has a good fit to a certain use case, and thus we believe all approaches have a good justification for co-existence.

2) ARTIFACT IDENTIFICATION

Once we derive program representation, we can traverse the structure to locate various artifacts. These artifacts are either components or a lower level constructs such as annotation.

To derive components typically involves graph traversal while checking each vertex to certain proprieties. Most of the frameworks use descriptors that identify a component, and each component can be located through a single graph traversal (e.g., breadth-first search). It is also possible to perform single traversal to hash components or artifacts according to its type and make them available for fast localization.

The most common components that we recognized to be searched for are endpoints that identify the system API and the data entities that match the data schemes. Identification of these and other components in modern enterprise frameworks typically happens through annotation descriptors or supertype identification.

Endpoints are critical components because they identify the boundary between the system and the outside. On their own, they identify the black box entries to the system. Typically, endpoints are the location where processes start, and they also typically enforce security. They might be the weak entry points vulnerable to attacks. Broader system analysis might first identify endpoints and then consider that happens next through the analysis of the internal structures and calls.

The data entity represents the data with which the system operates. Data describe the static perspective of the system, define the domain and scope of the system. If we are looking for a data breach localization, data entity identification and its structure are the first points of interest. Besides, when we aim to utilize data structure for its UI representation, data entities might be the only components of interest.

Both endpoints and data entities can be recognized by introspection, source, and bytecode analysis. If we consider three-layered architecture, identifying these two types of components is essential. The middle components, such as services or repositories, can be easily identified as well.

However, when we are interested in more fine-grained details, introspection might become insufficient. It might work when identifying interface, component structure, and component API, but no internal details. For instance, when we look for a location in the code where we perform a remote procedure call, send a message to the message queue, or connect to a data source, we can only base on the source and bytecode analysis. It is similar to the identification of failure locations, exceptions, or logs message origins.

3) FLOW ANALYSIS

When we consider a broader analysis of the system, we typically find two strategies: Data-flow and control-flow analysis. In the previous text, we mentioned the identification of endpoints or data entities, which are typically the first step in these flow analyses.

Control-flow is typically considered for each endpoint by tracing the call-graphs of inner methods. Thus, we can easily derive a forest of trees and identify their overlap since code design tends to reuse method. This brings excellent benefits when looking for consistency errors across overlaps, as did in security analysis in [84].

Endpoints typically accept parameters, and these are transformed into data captured by the system. Tracing the data through the system, its constrain checks and conversions involve data-flow analysis. In the end, we know the system understands data formats defined by the data entities. Data entities carry other semantic information, such as constraints on fields or input validation. We know that data from the endpoint parameter can only be traced down to the data entity through the control-flow.

We recently observed that most control-flow approaches work great for explicit invocation, not for implicit invocation. For instance, is the method call is intercepted, an event is produced and observed, CFG might not reflect these. It is especially the case in enterprise systems that use these implicit-invocation control mechanisms often.

When dealing with data-flow or control-flow analysis, we might have a specific goal, and it can be transformed onto and formal specification [123].

4) PATTERN IDENTIFICATION

Another perspective of analysis is recurrent pattern identification. Patterns can identify a common error, vulnerabilities, or possibly wrong design. Besides, code clone detection looks for similar patterns [91].

Bug patterns are the main driver for many static analysis tools, including among many SpotBugs, and PMD. SpotBugs [87] (initially FingBugs) is a Java code tool seeded in the University of Maryland. It identifies common issues such as unclosed file handles, infinite loops, missing null pointer check, and many others. SpotBugs plugins [124] even identify 135 vulnerabilities, including random generators, hashes, HTTP headers, SSL, path traversal, various injections, certificates, XML parsing, input validation, leaks, cross-site scripting, deserialization, and many others. PMD [88] is another tool similar to SpotBugs but with support to multiple platforms. However, a common deficiency for these tools lacks support for a distributed system.

Similarly, code clone detection is not only an exact match but also a partial match, which makes the challenge more complicated and time demanding [91]. It, however, shares the challenge, the support for distributes systems. Another perspective considered in the code is code smells that indicated poor design [125]. Smells are not necessarily errors but lower the quality of the system design. smells.

5) METRICS

Static analysis has also been used to assess system quality. Typically, we can use various metrics [126]. Such metrics include the Number of Logical Lines of Code (LLOC), Depth of inheritance tree (DIT), Weighted method per class (WMC), Coupling between objects (CBO), Number of children (NOC), Number of methods (NOM), Lack of cohesion

among methods (LCOM), Response for class (RFC), Data abstraction coupling (DAC), McCabe Cyclomatic complexity (MVG), and many others. Such metrics can indicate an issue in component design or the interaction among components. Similar to code smells, metrics do not necessarily indicate an error but indicate that refactoring should be performed to improve maintainability and readability, which may eventually pay for the efforts.

6) TRANSFORMATIONS

The final perspective we consider includes transformations, reverse engineering, model derivation, and documentation extraction.

In particular, analysts do not understand the source code of enterprise systems and various perspectives, details, and documents that can be extracted from the code for them. Vice-versa, analysts may want to influence the system's business logic without developer involvement, and thus, the system needs to integrate DSL scripts comprehensible to analysts and interpretable within the enterprise system.

Code analysis enables documentation extraction; as mentioned earlier, it is fairly common to use the analysis to determine data scheme and endpoints, and proper transformation may turn it into UML models or other appropriate structures [80], [127]. DSL makes it suitable for analysts to focus on specific system domain, and since these languages are interpreted, such can be part of the GPL system logic. This, however, implies that to understand the system holistically through code analysis, these special DSL documents must be considered in the analysis, which is also the case for build and deployment configurations.

V. CODE ANALYSIS IN DISTRIBUTED ENTERPRISES

Modern systems target the scalability and move towards distribution, in particular to a cloud [26]. In practice, each module is likely to be developed in separate code repository [128]. With the insight on code analysis, the first approach to maintain code quality might be to process multiple repositories and combine the results. However, this is rarely possible, especially considering the MSA design [23], which is the current trend for cloud-computing and enterprise systems. Each MSA module defines its constraints, security roles, data scheme, business rules, etc. [3]. There is not a centralized perspective that exits in monolith systems or legacy Service-Oriented Architectures (SOA) involving Enterprise Service Bus (ESB) [23], [26]. Moreover, multiple languages or frameworks can be involved across different modules with different programming styles.

There are very few multi-repository quality assurance tools supporting a centralized dashboard across multiple distinct projects along with multi-language recognition. This tool is Fabric8-analytics [128], [129]. However, it does not combine the module-separated information and distributed knowledge into a single perspective to draw the global picture of the overall system. The global perspective here is the key since it would enable us to identify inconsistencies across modules, determine and optimize the distributed processes, and their performance. Access to a global perspective would allow us to be more efficient when testing the system, considering change impact, or finding overlapping functionality.

Typically all the responsibilities and rules are evident when we design the system from scratch, but since each module is managed and evolved by a separate team, the evolution might turn it a wrong direction and the preventive detection mechanisms indicating a problem are currently missing. Moreover, the current practice in system integration is that developers typically know other modules only from the high-level perspective and recognize some of its endpoints or API, but not the actual implementation. The situation is far more difficult as developers design and implement the MSA modules, but then pass them to SysOps who know to deploy modules to the cloud infrastructure but have no detailed idea of the particular module specifics to optimize the deployment.

We could continue to enumerate implied problems; however, where do we see a possible solution direction to all these kinds of problems? In code analysis. Nevertheless, there are multiple obstacles to applying code analysis to enterprise MSA-based solutions, and we identify these next in the following text along with our prototypical experiments.

A. VIRTUAL CENTRALIZED INFORMATION OVERLAY

We mentioned earlier that each MSA module defines its knowledge, including constraints, business rules, and data schemes. Each module operates in its bounded context rather than in a global context. Each module encapsulates the domain knowledge from other system modules, and thus, other modules have no internal knowledge besides its scope. However, we would like to derive the overall picture and use it to assist with consistency checking. Thus we can analyze each module code to identify its data schemes and combine them to derive what is called a context map [25].

However, combining data schemes from multiple modules is not trivial since the same data entity might have a different name and a subset of properties on one side or even inconsistent data types or field constraints. In our experimental prototype code analyzer, called Prophet,[1] we use text similarity algorithms to bind entities across modules. This brings a virtual overview to the canonical model, identifies data dependencies and coupling across modules. Next, it helps to identify which control-flow paths could be cross-checked across modules for consistencies. For instance, for each module, we can identify methods operating with the particular data entity and backtrace the control-flow to the endpoints. This way, we identify all intra- and inter-module endpoints that are dependent on the particular data entity and its constraints. Currently, we experimented with Java and Python heterogeneous projects involving both source code and bytecode, and our online website of Prophet demonstrates

[1] Prophet is available online at https://cloudhubs.ecs.baylor.edu/prophet/#/ with open sources at https://github.com/cloudhubs

the applications on a third-party MSA testbed introduced in [130].

Another perspective that is worth considering involves call graphs. It identifies all system endpoints along with inter-module calls towards these endpoints. This enables additional tracing of control-flow and possible inter-module dependencies for consistency checking. Even though the precise identification of interacting modules can be resolved at runtime [131], [132], static possible module interaction is sufficient as it reveals possible interaction combinations that could realistically happen upon the deployment. Prophet[1] considers this approach on the same MSA testbed [130].

B. ACCESS TO CONTAINERS

While the information centralizing approaches from the previous subsection work well on projects deployed locally, in a practical setting, we need to access the modules deployed in a cloud infrastructure [27] or a service-mesh [29]. Such modules are typically wrapped into a container, e.g., Docker [30], [31]. Thus, to apply the approach for the direct benefit of SysOps or deployed systems, it is necessary to access containers. Here multiple libraries exist for monitoring purposes [41] and provide the necessary access along with the ability to instrument deployed systems.

Since in Prophet, we considered information extraction from both the source code and bytecode, it can be directly integrated into frameworks building on the top of Docker using the existing libraries for [41]. With this accomplishment, it becomes possible to fabricate a centralized system overview to SysOps.

C. DEPLOYMENT/BUILD CONFIGURATION ANALYSIS

Source code analysis cannot determine broad details about deployment and build dependencies, especially when we wrap modules in containers. Typically, enterprise software projects contain build-files and deployment configuration files. These files contain details that enable automated containerization, and this information is beneficial to analyze.

For instance, Ibrahim *et al.* [133] use a project's Docker files. In their research, they analyze these files to derive system topology. In particular, they extract information from Docker Compose files to generate an attack graph showing how a security breach of containers could propagate through a microservice mesh. Analysis of deployment descriptors allows creating, to some extent, the interdependence of modules. However, in their approach, they do not extend the analysis to the source code. Thus, it cannot identify security flaws in the program deployed in the containers, only flaws with the images themselves.

Using deployment artifacts for containers, particularly Docker Compose files and build scripts, support identifying the architecture of a microservice system [134]. It can identify remote endpoints and services and dependencies. In [135], authors used docker files for architectural reconstruction and highlighted that these files represent valuable analysis targets.

If we only analyzed deployment and build files, we would know nothing about module internal details, features, and specifics, including details about utilized data, control flow, restrictions constraints, etc. Therefore, a microservice system's general structure can be recreated with static code analysis if deployment descriptors are involved. Information extracted from the individual microservices through code analysis can be augmented with the architectural information from deployment artifacts, which inherently describes the system as a whole. The individual and overall information can be combined to create a more useful analysis. This can be helpful, especially in distributed environments.

In addition to generating attack graphs and reconstructing the system's architecture, deployment scripts can be analyzed to detect possible security violations. Containers, unlike VMs, share the kernel with the host which provides lightweight virtualization [136]. However, this might lead to potential kernel security issues such as privilege escalation [136] facilitated by additional permissions in deployment scripts. While these permissions may be essential for some containers, end-users need to be cautious while deploying third party containers to avoid unintended approval of additional permissions. By investigating deployment scripts it is possible to detect and mitigate these security issues, for instance, there exists an approach [137] to eliminate privilege escalation by analyzing Dockerfiles and Kubernetes pod specifications.

D. ON DISTRIBUTED SYSTEM VERIFICATION

The ability to fabricate a virtual centralized system overview opens the distributed MSA systems to broader verification. For instance, one can consider consistency checking along the call-graphs for data constraints. If two modules apply different constraints, the inconsistency is reported. One can also trace security consistencies. Enterprise systems typically apply Role-Based Access Control (RBAC) over the endpoints [16], [82], [83], [138], [139] to protect the system from intruders. Each user is then assigned a particular access role that activates in a particular context. Nowadays, the best practice [16], [83], [139] is to annotate endpoints with allowed user roles.

In [84], we considered consistency checking in a single module involving control-flow graphs. Each endpoint was traversed down to data persistence. Throughout the traversal, it assigned the intermediate methods a virtual role identified by a particular endpoint. Since systems are implemented using structural decomposition, the methods and components are often reused. Thus, we could notice if there existed a method or resource on two or more paths from various endpoints with a different security policy and report the inconsistency. This leads to multiple possible outcomes, such as inconsistent roles, inconsistent role hierarchy, method exposed to the public, unknown role, or even inconsistent security policy applied to a particular resource or method, e.g., persistence or access involving a particular data entity.

With the centralizing overview fabrication, this approach can be applied globally. This, however, brings one challenge that requires role mapping. Since modules apply their knowledge, one cannot assume that the same role name in distinct modules has an equivalent meaning and impact. The role access-resolver needs to be taken into account, or a manual role mapping can be supplied.

E. FINDING CODE SMELLS

In the [125], the authors identify code smells in MSA. These smells include improper module interaction, modules with too many responsibilities, and often a misunderstanding of the architecture. For instance, there might be SOA-based ESB-like modules to pass messages between modules. There might be too many standards involved across discrete teams of developers when a single standard should be established for consistency across the modules.

On the communication, there might exist a wrong cut to layers. There might be a missing manager for connections between MSA modules (API gateway), and direct communication is involved. For instance, there might be hard-coded endpoints with IP addresses and ports.

Regarding the development and design process practice, there might exist no API versioning, or there are too many MSA modules for small purposes. Another issue to find is shared persistence using the same database rather than involving individual data storage. Similarly, inappropriate service intimacy might exist when one module access encapsulated data of another service, which is similar to shared libraries or a cyclic call dependency.

These smells can be detected manually, requiring assessment and a basic understanding of the system demanding considerable effort. However, with code analysis instruments, these smells are natural to discover almost instantly with no previous system knowledge. Such internal audit can be part of a broader quality evaluation, regular iterative milestone deliverable audit, etc. In particular, intra-module recognition of enterprise components allows us to recognize abstract structures, and draw the interaction patterns. Together with inter-module communication and canonical data model, we can further recognize high-level interconnection, compare large to small module responsibilities, their dependencies, used resources, internal knowledge, or involved bad coding practice. We have developed such a tool for MSA systems called MSANose.[2] It can detect all eleven bad smells defined by [125].

VI. THE ASPECT-ORIENTED PERSPECTIVE

One possible perspective we could look into distributed systems is the perspective of AOP [76]. Each module consists of particular concerns—for instance, persistence, security, data constraints with input validation, or business rules. In monoliths like systems or individual, distributed modules, we aim to involve proper design that separates concerns [140]. Modern development frameworks enable reasonable concern separation through method interception [141] and annotation descriptors.

However, there is no reasonable concern separation when it comes to the overall distributed system. Each module redefines the concerns. Moreover, no centralized concern view exists. One has to assess each module to combine the scattered concern to get the whole picture. If we consider the module assessment as one stage and information consolidation as the second, centralizing stage, we could draw a similarity with distributed AOP [26]. The code analysis can be utilized to assess modules for concerns, and thus, be the first part of the distributed aspect weaving. Involving such analysis, we can obtain module-specific details and use them for weaving into a global concern perspective.

Great example is security concern [16], [83]. Here we define an access role, bind it to an annotation intended to augment an endpoint, and specify authorizer for the particular role in a given context. If we want to authorize a particular method, we annotate it with given role annotation. This leads to a rather clean separation of concerns. The role annotation acts as a join point. The authorizer consists of the authorization logic reusable for all locations in the code space using given role annotation. We can archive a rather solid separation of concerns in single code space. However, the MSA design direction goes in a reverse direction. Since each module is self-contained, it contains its concern definitions. This means that if one module uses a certain access policy, another module cannot reuse it. It has to reinvent it.

Separating the concern to a third component would, however, be a code smell leading back to SOA and ESB, even though it is rather a common research direction in distributed AOP [26], [142]–[157]. Thus, the most straight forward direction, to maintain module independence in MSA, is to develop replicas of concerns for each module. These then scatter across the overall system, making it difficult to see the overall concern span or to maintain consistency. Existing distributed AOP techniques [26], [142]–[157] target remote point-cuts, which leads to MSA any-pattern [125]. However, when we look at the original AOP intention introduced in [76], there are multiple phases in the AOP weaving process; thus, if we consider building a distributed overlay join point model involving individual modules, we could combine it to receive a virtual view on a particular concern. From there, we can easily verify the consistency of repeated concerns definitions.

When we consider the security concern mentioned earlier in the previous section, we can detail the assessed parts. First, the module-specific concern is identified by the access role authorizers identifying user roles. Next, the role annotations indicate the join points with the code logic, typically endpoints. We can further perform traversal of the control-flow from the endpoints, which will derive the overlapping paths. Taking into account the endpoint security policies, we can identify conflicts. Since we can combine modules either through the bounded-context merge into a canonical

[2]https://github.com/cloudhubs/msa-nose

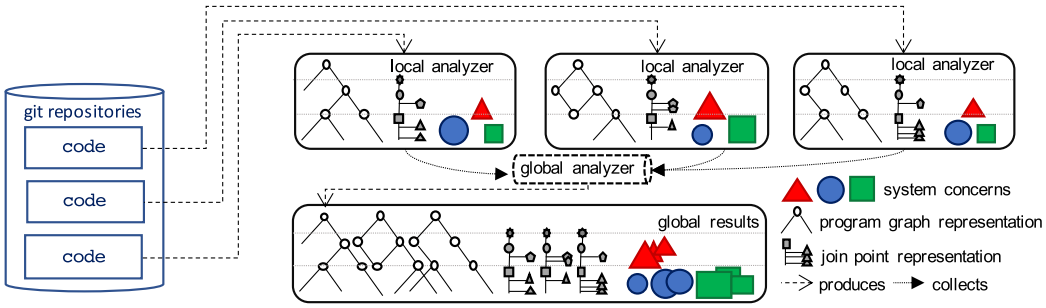

FIGURE 1. Code analysis use to centralize concerns scattered across distributed system modules.

model or through the module remote call interaction, we can combine both inter- and intra- module traversal paths to discover fundamental consistency issues that are hard to discover otherwise. However, the authorizer merging is currently the subject of our research and experiments.

This approach is not concern-specific, and we used security as a concern example. We can centralize data schemes, constraints, and input validation. Moreover, if the business rules are captured in readable form, as suggested in [80], [81] one can easily combine and formally verify the consistency of business rules from the overall system.

Besides, this process of concern centralization can provide access to the up-to-date system documentation, which is currently difficult to access since modules change often, and the documentation easily gets out-dated. Similarly, there is a notable delay in co-evolution between the system source code and tests [158], which further facilitates the introduction of errors, not to mention that current integration test approaches resort to only consider the high-level module interaction [43], [44] due to the complexity and difficulties related to accessing system details. We highlight the centralization for concerns scattered across distributed system modules in Figure 1.

VII. CONCLUSION

This study presents selected challenges in enterprise systems and microservices. The manuscript intends to motivate the community to broaden the research interest in code analysis targeting these systems. Many of the highlighted problems remain the bottleneck in current mainstream development. We believe that code analysis considering the enterprise system development practice can provide an efficient instrument for what is currently tedious and error-prone manual work. In particular, we demonstrate two techniques we use to merge microservice modules into a virtual overlay perspective, allowing us to apply various types of verification above the scope of a single code-centered module. The approach is platform-independent since current development standards remain similar across different platforms; however, we only verified this on Java and Python platforms. We presented possible categorization of code analysis target directions. Besides, we presented an AOP perspective recognizing the issue of scattered concerns in distributed system modules.

In the appendix, we present provides a list of open-source tools that might be useful to researchers willing to challenge this field. Furthermore, since one may face the difficulty of finding useful benchmarks for their approach evaluation, we share a list of microservice applications previously used by researchers for evaluation of their approaches.

A. FUTURE CHALLENGES AND OPPORTUNITIES

Code analysis can bring automation to various enterprise tasks, including verification, code to code transformation, documentation, or information extraction for non-developers that understand the code. Code analysis provides excellent opportunities for testing and can help with the derivation of test-cases. We see an opportunity to reduce the testing time by employing code analysis. Besides, useful reports can be provided to developers, SysOps, or other users on code quality, code clones, smells, bug patterns found in the code or metrics for audits or quality analysis. However, many other opportunities come with challenges.

Among many other challenges in the field, one could consider log analysis of interacting modules. With access to the system code, the log can provide essential information to recognize the runtime issues and anomalies, leading to possible near-future system failure, vulnerability, or an ongoing attack. Moreover, it has the potential to straighten quality assurance practice. Log to code mapping or generally program slicing [159] is another active field of research where more research is necessary.

Code analysis can be directly connected with repository mining [115], [118]–[120], where a new code repository push is assessed from the module perspective. It has the advantage of providing quick feedback to the responsible developer or quality assurance specialist [125]. This all can happen even without module deployment to the test environment. However, for the context of enterprise distributed systems, it is necessary to assess the impact of changes from the holistic system perspective. With such ability, we would be able to address new types of errors that are hard to find manually by distributes system assessments. The evident challenge is to derive the holistic system view automatically [135], given the circumstances of possible heterogeneity across modules.

TABLE 1. Available open source tools that can be utilized in this research.

Name	Details	URL
MetaWidget	Library to generate UI from backend code	http://www.metawidget.org/
AspectFaces	Library to generate context-aware UI from backend code	https://bitbucket.org/CodingCrayons/aspectfaces
ApacheISIS (NakedObjects)	Library to generate UI from backend code	http://isis.apache.org/
MSANose	MSA bad smell detection too	https://github.com/cloudhubs/msa-nose
Prophet	Online tool for microservice overlay derivation	https://cloudhubs.ecs.baylor.edu/prophet
Prophet Utils	Library to assess microservice solutions and derive module overlays	https://github.com/cloudhubs/prophet-utils
Rest API Discovery	Library to detect the REST endpoints and clients (API calls)	https://github.com/cloudhubs/rad-source
Bounded Context	Library to generate bounded context from a microservice module	https://github.com/cloudhubs/bounded-context
JParser	Library to analyze enterprise concerns from source code	https://github.com/cloudhubs/jparser
Global weaver	Library to analyze security policies based on Java Security API [16], [110]	https://bitbucket.org/cilab/global-weaver/src/master/

With an automated approach to derive the overall system perspective, SysOps could benefit from a better understanding of the underlying modules with detailed knowledge of their salient aspects. This would bring great opportunities. For instance, one could quickly extract up-to-date documentation for the overall system [135]. SysOps are not developers, so their knowledge about particular modules is limited to the assessment of deployment descriptors, but important module specifics can be missed, which brings another opportunity. Moreover, such an approach would bring benefits from the system quality perspective allowing for distributed code smell detection [160] or distributed code clone detection [91], [92]. However, automated derivation of the overall distributed system perspective remains a challenge and deserves attention.

If we could overcome the current challenge of lacking tools extracting the knowledge of module-specific details, fine-modularity of inter-module dependencies, and specific concern scattering, besides the new mechanisms to assess systems, we gain many new opportunities for further research. We could develop novel monitoring and debugging tools basing on concern-scattering. This could lead to concern-inversion debugging, which would enable system debugging/logging based on a selected concern in the overall system no matter where it is defined rather than based on explicit knowledge of lines and components where a specific module defines the assessed concern, which is the current practice and something SysOps cannot adequately perform.

Furthermore, access to an automatically derived centralized perspective of a distributed system could provide an oracle to testers, analysts, and software architects. It would simplify system testability and lead beyond a basic high-level module interaction evaluation used these days [43], [44]. For instance, a black-box approach when checking on process flow [161] or message exchange [162]–[164] might provide limited results as opposed to a white-box approach. However, currently, the white-box testing demands efforts and costs. With an automated approach to derive the global perspectives of the distributed systems, we could quickly access details; the tests could be designed from the white-box perspective without increased efforts.

Similar concepts proven to be perspective in the field of Combinatorial [165] and Constrained [166] Interaction Testing (CIT). Studies suggest good capability to generate CIT-based test cases from the actual state of code [167]–[169], and this concept is mostly applicable in automated generation of unit tests, where also alternatives have been suggested [170], [171]. Not only is CIT employed in automated generation of tests from source code, as another example, the mutation testing approach can also be used [172]. The capability to generate unit tests from the source code is potentially applicable in the case of distributed systems as well, as no principal differences in the context may impact such an application. Furthermore, automated test case generation could better target selected system aspects or the global system characteristics rather than optimizing the per-module perspective.

Another direction where enterprise code analysis can be instrumental involves penetration testing or just pen-testing. The span of pen-testing ranges from the network and operating system security through application security, including code analysis. We showed one interesting approach related to attack graph generation from container deployment descriptor [133] where container vulnerabilities and their propagation across modules can be assessed. Assessment of found vulnerabilities helps to uncover system weaknesses and flaws. Considering the current testing practice of distributed systems, one can expect rather poor results with respect to weak security points [176], [177]. Various analysis tools [87], [88], [124], [178]–[181] help with pen-testing. They involve system code and pattern matching or examining the running state to indicate issues. The automation involving code analysis can range from encryption detection, an indication of back doors (hardcoded roles and passwords), unhandled situations, exceptions, or incorrect settings (e.g., limiting cross-site scripting). Existing tools typically address the top OWASP vulnerabilities [182]. However, the distributed perspective remains an open challenge [128].

TABLE 2. Identified microservice benchmarks.

Name	Language	URL	Ref
Acme air	NodeJS	https://github.com/acmeair/acmeair-nodejs	[173]
Apolo	Java	https://github.com/ctripcorp/apollo	[173]
Atsea Sample Shop App	Java	https://github.com/dockersamples/atsea-sample-shop-app	[133]
EnterprisePlanner	C#	https://github.com/gfawcett22/EnterprisePlanner	[173]
eShop	C#	https://github.com/dotnet-architecture/eShopOnContainers	[173]
Freddy's bbq joint	Java/JavaScript	https://github.com/william-tran/freddys-bbq	[173]
Gizmo	Go	https://github.com/nytimes/gizmo	[173]
JavaEE demo	Java	https://github.com/dockersamples/javaee-demo	[133]
Magda	JavaScript/Scala	https://github.com/magda-io/magda	[173]
Microservices Reference	C#	https://github.com/mspnp/microservices-reference-implementation	[173]
Netflix OSS	Java	https://github.com/Oreste-Luci/netflix-oss-example	[133]
Photo uploader	JavaScript	https://github.com/nginxinc/mra-ingenious	[173]
PHPMailer and Samba	PHP	https://github.com/opsxcq/	[133]
Piggy Metrics	Java	https://github.com/sqshq/piggyMetrics	[173]
Pitstop	C#	https://github.com/EdwinVW/pitstop	[173]
Share bike	Java	https://github.com/JoeCao/qbike	[173]
SiteWhere	Java	https://github.com/sitewhere/sitewhere	[173]
Sock Shop	Java/Go/NodeJS	https://github.com/microservices-demo/microservices-demo	[173]
Texas Teach Training	Java	https://github.com/cloudhubs/tms2020/	[174]
Train Ticket Benchmark	Java	https://github.com/FudanSELab/train-ticket	[130]
Vehicle tracking	C#	https://github.com/mohamed-abdo/vehicle-tracking-microservices	[173]
Warehouse microservice	Java	https://github.com/HieJulia/warehouse-microservice	[173]
WeText	C#	https://github.com/daxnet/we-text	[173]
Other benchmarks to be found at the curated list of microservice projects at https://github.com/davidetaibi/Microservices_Project_List			[175]

This kind of testing involving static analysis is sometimes referred to as Static Application Security Testing (SAST) [181]. It can be applied in early application development without the time constraints and delayed feedback of manual testing, which is a great advantage. Pen-testing can use details exposed or analyzed by SAST, and using both methods increases productivity. SAST is likely to reduce the number of security issues and vulnerabilities.

Still, it does not consider the existence of enterprise development standards and commonly used frameworks for application design. Inclusion of these could improve the assessment precision and quality. Also, SAST functions for modules in isolation. It does not support a comprehensive assessment of distributed systems as if they were a monolith solution without module boundaries and possible heterogeneity in modules. With static analysis from the holistic perspective, the module interdependence could be detected. Existing tools could take the opportunity to utilize methods to for interdependence detection to better vulnerability propagation [133] or to focus on possible fragile parts of the communication chain. Besides, such a perspective would open new opportunities to perform RBAC policy cross-checking of interacting methods [84] to identify consistency errors.

To summarize, current trends in enterprise systems open many interesting challenges to the research communities to fill the open gaps. Utilizing code analysis of enterprise systems can bring opportunities to improve the efficiency of the processes, speed them up, address the current extended efforts or provide new perspectives so far addressed manually. Code analysis of enterprise systems is an auspicious direction to address the current challenges to provide practical opportunities. We believe it deserves a much broader research interest. To support our peers in this research direction, we share our open-source tools that can be used and extended to perform further experiments and tool development. Apart from this, we also share a list of microservice-based applications previously used as evaluation testbeds.

APPENDIX A
REFERENCE OPEN-SOURCE TOOLS

In the manuscript are mentioned various challenges. To some, we provided references to tools trying to address them. Table 1 provides summary references to related open source tools. In the first column it provides the tool name, the next column explains what is the tools meant for and the third column contains HTTP link to the tool.

APPENDIX B
LIST OF MICROSERVICE BENCHMARKS

Researchers typically need to test their approaches [183]. For a long time there was a lack of available microservice benchmarks [173] and references are often hard to find. For these reasons we include a list of benchmarks we identified across literature and sources we assessed [130], [133], [173]–[175], [184]. Table 2 provides a reasonable list of microservice benchmark references which can be used to test new tools and approaches. In the first column it provides the benchmark name, the next column lists the programming language, the third column contains HTTP link to the tool and the finally a reference to the citing paper.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.

[2] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.

[3] K. Finnigan, *Enterprise Java Microservices*. Shelter Island, NY, USA: Manning Publications, 2018. [Online]. Available: https://books.google.com/books?id=KaSNswEACAAJ

[4] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2012.

[5] *OMG, Unified Modeling Language (OMG UML), Infrastructure (Version 2.2)*, Object Manage. Group (OMG), Needham, MA, USA, Jan. 2009. [Online]. Available: https://www.omg.org/spec/UML/2.2/About-UML/

[6] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2004.

[7] A. Tijms. (2019). *Jakarta EE 9—Release Plan*. Accessed: Jul. 16, 2020. [Online]. Available: https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9ReleasePlan

[8] (2015). *Primefaces User Interface Framework*. Accessed: Jul. 16, 2020. [Online]. Available: http://primefaces.org

[9] (Apr. 2015). *AngularJS Documentation*. Accessed: Jul. 16, 2020. [Online]. Available: http://angularjs.org

[10] T. Abel, *ReactJS: Become a Professional in Web App Development*. Scotts Valley, CA, USA: CreateSpace Independent Publishing Platform, 2016.

[11] L. DeMichiel and W. Shannon. (2016). *JSR 366: Java Platform, Enterprise Edition 8 Spec*. Accessed: Jul. 16, 2020. [Online]. Available: https://jcp.org/en/jsr/detail?id=342

[12] L. DeMichiel. (Nov. 2009). *JSR 317: JavaTM Persistence API, Version 2.0*. Accessed: Jul. 16, 2020. [Online]. Available: http://jcp.org/en/jsr/detail?id=317

[13] E. Bernard. (Nov. 2009). *JSR 303: Bean Validation*. Accessed: Jul. 16, 2020. [Online]. Available: http://jcp.org/en/jsr/detail?id=303

[14] G. King. (2017). *JSR 299: Contexts and Dependency Injection for the Java EE Platform*. Accessed: Jul. 16, 2020. [Online]. Available: https://jcp.org/en/jsr/detail?id=299

[15] R. Grigoriadi. (2017). *JSR 222: Java Architecture for XML Binding (JAXB)*. Accessed: Jul. 16, 2020. [Online]. Available: https://jcp.org/en/jsr/detail?id=222

[16] W. Hopkins. (Nov. 2009). *JSR 375: Java EE Security API*. Accessed: Jul. 16, 2020. [Online]. Available: https://jcp.org/en/jsr/detail?id=375

[17] Microsoft. (2019). *Net Entity Framework Documentation*. Accessed: Jul. 16, 2020. [Online]. Available: https://docs.microsoft.com/en-us/ef/

[18] M. Makai. (2019). *Object-Relational Mappers (ORMS)*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.fullstackpython.com/object-relational-mappers-orms.html

[19] J. H. Wage. (2019). *The Doctrine Project (PHP)*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.doctrine-project.org

[20] Laravel, LLC. (2019). *Laravel PHP Framework*. Accessed: Jul. 16, 2020. [Online]. Available: https://laravel.com/docs/5.8/authorization

[21] L. Jungmann. (2020). *JSR: Java Specification Requests JSR 338: JavaTM Persistence 2.2*. Accessed: Jul. 16, 2020. [Online]. Available: https://jcp.org/en/jsr/detail?id=338

[22] H. Buelow, M. Deb, J. Kasi, D. LHer, and P. Palvankar, *Getting Started With Oracle SOA Suite 11G R1 A Hands-On Tutorial*. Birmingham, U.K.: Packt, 2009.

[23] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *ACM SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, Jan. 2018, doi: 10.1145/3183628.3183631.

[24] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing—A systematic mapping study," *J. Syst. Softw.*, vol. 126, pp. 1–16, Apr. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217300018

[25] E. Wolff, *Microservices: Flexible Software Architectures*. Scotts Valley, CA, USA: CreateSpace Independent Publishing Platform, 2016. [Online]. Available: https://books.google.com/books?id=X7YzjwEACAAJ

[26] T. Cerny, "Aspect-oriented challenges in system integration with microservices, SOA and IoT," *Enterprise Inf. Syst.*, vol. 13, no. 4, pp. 467–489, Apr. 2019, doi: 10.1080/17517575.2018.1462406.

[27] Google. (2018). *Kubernetes: Automated Container Deployment, Scaling, and Management*. [Online]. Available: https://kubernetes.io

[28] R. Hat. (2018). *OpenShift: Container Application Platform*. [Online]. Available: https://www.openshift.com

[29] ISTIO. (2018). *ISTIO, An Open Platform to Connect, Manage, and Secure Microservices*. Accessed: Jul. 16, 2020. [Online]. Available: https://istio.io

[30] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering With Swarm*. Birmingham, U.K.: Packt, 2017.

[31] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[32] NGINX. (Nov. 2015). *The Future of Application Development and Delivery Is Now Containers and Microservices Are Hitting the Mainstream*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.nginx.com/resources/library/app-dev-survey/

[33] (Jun. 2018). *Microservices Architecture Market Research Report-Global Forecast 2023*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.marketresearchfuture.com/reports/microservices-architecture-market-3149

[34] (Nov. 2017). *Microservice Architecture Market, 2023*. Accessed: Jul. 16, 2020. [Online]. Available: https://globenewswire.com/news-release/2017/11/03/1174391/0/en/Microservice-Architecture-Market-2023.html

[35] B. Gracely. (2018). *Understanding Service Meshes*. [Online]. Available: https://blog.openshift.com/podcast-podctl-basics-understanding-service-meshes/

[36] P. Belagatti. (May 2019). *How Big Companies are Using Kubernetes*. Accessed: Jul. 16, 2020. [Online]. Available: https://jaxenter.com/big-companies-using-kubernetes-159007.html

[37] Datadog. (Dec. 2018). *8 Emerging Trends in Container Orchestration*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.datadoghq.com/container-orchestration/

[38] CNCF Survey. (Aug. 2019). *CNCF Survey 2019: Deployments are Getting Larger as Cloud Native Adoption Becomes Mainstream*. Accessed: Mar. 10, 2020. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf

[39] Kaitlyn Barnard. (Aug. 2018). *CNCF Survey: Use of Cloud Native Technologies in Production Has Grown Over 200%*. Accessed: Nov. 10, 2019. [Online]. Available: https://www.cncf.io/blog/2018/08/29/cncf-survey-use-of-cloud-native-technologies-in-production-has-grown-over-200-percent/

[40] (2018). *Knative: Building Blocks That Simplify How You Deploy and Run Functions a Top Kubernetes and Istio. On Any Cloud*. [Online]. Available: https://www.openshift.com

[41] Prometheus Authors. (2019). *Prometheus Monitoring*. Accessed: Nov. 10, 2019. [Online]. Available: https://prometheus.io/

[42] Grafana Labs. (2019). *Grafana: The Open Observability Platform Monitoring*. Accessed: Nov. 10, 2019. [Online]. Available: https://grafana.com/

[43] M. Camilli, C. Bellettini, and L. Capra, "Design-time to run-time verification of microservices based applications," in *Software Engineering and Formal Methods*, A. Cerone and M. Roveri, Eds. Cham, Switzerland: Springer, 2018, pp. 168–173.

[44] M. Camilli, C. Bellettini, L. Capra, and M. Monga, "A formal framework for specifying and verifying microservices based process flows," in *Proc. Int. Conf. Softw. Eng. Formal Methods*. Cham, Switzerland: Springer, 2017, pp. 187–202.

[45] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura, "A metaprogramming framework for formal verification," in *Proc. ACM Program. Lang. (ICFP)*, vol. 1, Aug. 2017, pp. 34:1–34:29, doi: 10.1145/3110278.

[46] A. Chlipala, "The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 391–402, Nov. 2013, doi: 10.1145/2544174.2500592.

[47] H. Cho, "Using metaprogramming to implement a testing framework," in *Proc. 47th Annu. Southeast Regional Conf. (ACM-SE)*, New York, NY, USA, 2009, p. 55, doi: 10.1145/1566445.1566519.

[48] P. E. Papotti, A. Prado, and W. L. de Souza, "Reducing time and effort in legacy systems reengineering to mdd using metaprogramming," in *Proc. ACM Res. Appl. Comput. Symp.*, Oct. 2012, pp. 348–355.

[49] V. R. L. Mendonca, C. L. Rodrigues, F. A. A. MN Soares, and A. M. R. Vincenzi, "Static analysis techniques and tools: A systematic mapping study," in *Proc. 8th Int. Conf. Softw. Eng. Adv. (ICSEA)*, 2013, pp. 72–78.

[50] J. C. B. Ribeiro, F. F. de Vega, and M. Zenha-Rela, "Using dynamic analysis of java bytecode for evolutionary object-oriented unit testing," in *Proc. 25th Brazilian Symp. Comput. Netw. Distrib. Syst. (SBRC)*, 2007, pp. 143–156.
[51] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla, "Verification of java bytecode using analysis and transformation of logic programs," in *Practical Aspects of Declarative Languages*, M. Hanus, Ed. Berlin, Germany: Springer, 2007, pp. 124–139.
[52] I. Keivanloo, C. K. Roy, and J. Rilling, "SeByte: Scalable clone and similarity search for bytecode," *Sci. Comput. Program.*, vol. 95, pp. 426–444, Dec. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313002773
[53] I. Keivanloo, C. K. Roy, and J. Rilling, "Java bytecode clone detection via relaxation on code fingerprint and semantic Web reasoning," in *Proc. 6th Int. Workshop Softw. Clones (IWSC)*, Piscataway, NJ, USA, Jun. 2012, pp. 36–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=2664398.2664404
[54] G. Chatley, S. Kaur, and B. Sohal, "Software clone detection: A review," *Int. J. Control Theory Appl.*, vol. 9, pp. 555–563, Jan. 2016.
[55] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill, "Identifying and documenting false positive patterns generated by static code analysis tools," in *Proc. 4th Int. Workshop Softw. Eng. Res. Ind. Pract. (SER&IP)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 55–61, doi: 10.1109/SER-IP.2017.20.
[56] F. Elberzhager, J. Münch, and V. T. N. Nha, "A systematic mapping study on the combination of static and dynamic quality assurance techniques," *Inf. Softw. Technol.*, vol. 54, no. 1, pp. 1–15, Jan. 2012.
[57] A. G. Bardas, "Static code analysis," *J. Inf. Syst. Oper. Manage.*, vol. 4, no. 2, pp. 99–107, 2010.
[58] I. J. Davis and M. W. Godfrey, "From whence it came: Detecting source code clones by analyzing assembler," in *Proc. 17th Work. Conf. Reverse Eng.*, Oct. 2010, pp. 242–246.
[59] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Inf. Softw. Technol.*, vol. 55, no. 7, pp. 1165–1199, Jul. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000323
[60] R. Kennard and J. Leaney, "Towards a general purpose architecture for UI generation," *J. Syst. Softw.*, vol. 83, no. 10, pp. 1896–1906, Oct. 2010, doi: 10.1016/j.jss.2010.05.079.
[61] R. Kennard, E. Edmonds, and J. Leaney, "Separation anxiety: Stresses of developing a modern day separable user interface," in *Proc. 2nd Conf. Hum. Syst. Interact.* Piscataway, NJ, USA: IEEE Press, May 2009, pp. 225–232. [Online]. Available: http://portal.acm.org/citation.cfm?id=1689359.1689399
[62] T. Cerny, M. Macik, M. Donahoo, and J. Janousek, "On distributed concern delivery in user interface design," *Comput. Sci. Inf. Syst.*, vol. 12, no. 2, pp. 655–681, 2015.
[63] T. Cerny, K. Cemus, M. J. Donahoo, and E. Song, "Aspect-driven, data-reflective and context-aware user interfaces design," *Appl. Comput. Rev.*, vol. 13, no. 4, pp. 53–65, 2013.
[64] T. Cerny and M. J. Donahoo, "On separation of platform-independent particles in user interfaces," *Cluster Comput.*, vol. 18, no. 3, pp. 1215–1228, Sep. 2015, doi: 10.1007/s10586-015-0471-7.
[65] T. Cerny, M. Trnka, and M. J. Donahoo, "Towards shared security through distributed separation of concerns," in *Proc. Int. Conf. Res. Adapt. Convergent Syst. (RACS)*, New York, NY, USA, 2016, pp. 169–172, doi: 10.1145/2987386.2987394.
[66] T. Cerny and E. Song, "Model-driven rich form generation," *Int. Inf.*, vol. 15, no. 7, pp. 2695–2714, 2012.
[67] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY, USA: Wiley, 1996.
[68] I. R. Forman and N. Forman, *Java Reflection in Action* (In Action Series). Greenwich, CT, USA: Manning, 2004.
[69] A. Torres, R. Galante, and M. S. Pimenta, "Towards a UML profile for model-driven object-relational mapping," in *Proc. 23rd Brazilian Symp. Softw. Eng.*, Oct. 2009, pp. 94–103.
[70] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.
[71] C. Bauer and G. King, *Hibernate Action* (In Action Series). Greenwich, CT, USA: Manning, 2004.
[72] R. Kennard and R. Steele, "Application of software mining to automatic user interface generation," in *Proc. Int. Conf. Softw. Methods Tools*. Amsterdam, The Netherlands: IOS Press, 2008.
[73] T. Cerny and E. Song, "A profile approach to using UML models for rich form generation," in *Proc. Int. Conf. Inf. Sci. Appl.*, 2010, pp. 1–8.
[74] R. Pawson and R. Matthews, "Naked objects: A technique for designing more expressive systems," *SIGPLAN Not.*, vol. 36, no. 12, pp. 61–67, 2001.
[75] L. DeMichiel and M. Keith. (May 2006). *JSR 220: Enterprise Javabeans Version 3.0. Java Persistence API*. Accessed: Jul. 16, 2020. [Online]. Available: http://jcp.org/en/jsr/detail?id=220
[76] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. Object-Oriented Program. (ECOOP)*, 1997, pp. 220–242.
[77] R. Laddad, *AspectJ in Action: Enterprise AOP With Spring Applications*, 2nd ed. Greenwich, CT, USA: Manning, 2009.
[78] T. Cerny and M. J. Donahoo, "On energy impact of Web user interface approaches," *Cluster Comput.*, vol. 19, no. 4, pp. 1853–1863, Dec. 2016.
[79] M. Bali, *Drools JBoss Rules 5.0 Developer's Guide*. Birmingham, U.K.: Packt, 2009.
[80] K. Cemus, F. Klimes, and T. Cerny, "Aspect-driven context-aware services," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Sep. 2017, pp. 1307–1314.
[81] K. Cemus, F. Klimes, O. Kratochvil, and T. Cerny, "Separation of concerns for distributed cross-platform context-aware user interfaces," *Cluster Comput.*, vol. 20, no. 3, pp. 2355–2362, Sep. 2017, doi: 10.1007/s10586-017-0794-7.
[82] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-Based Access Control*. Norwood, MA, USA: Artech House, Inc., 2003.
[83] Pivotal. (2019). *Spring Security*. Accessed: Jul. 16, 2020. [Online]. Available: https://spring.io/projects/spring-security
[84] A. Walker, J. Svacina, J. Simmons, and T. Cerny, "On automated role-based access control assessment in enterprise systems," in *Information Science and Applications 2019*. Cham, Switzerland: Springer, Dec. 2019.
[85] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.
[86] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, 2016, pp. 1157–1168, doi: 10.1145/2884781.2884877.
[87] 2019. *Spotbugs*. Accessed: Jul. 16, 2020. [Online]. Available: https://spotbugs.github.io
[88] (2019). *PMD: An Extensible Cross-Language Static Code Analyzer*. Accessed: Jul. 16, 2020. [Online]. Available: https://pmd.github.io
[89] B. Pugh. (2015). *Findbugs*. Accessed: Jul. 16, 2020. [Online]. Available: http://findbugs.sourceforge.net
[90] J. Svacina, J. Simmons, and T. Cerny, "Semantic code clone detection for enterprise applications," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 1–3.
[91] A. Walker, T. Cerny, and E. Song, "Open-source tools and benchmarks for code-clone detection: Past, present, and future trends," *SIGAPP Appl. Comput. Rev.*, vol. 19, no. 4, p. 28–39, Jan. 2020, doi: 10.1145/3381307.3381310.
[92] A. Walker and T. Cerny, "On cloud computing infrastructure for existing code-clone detection algorithms," *ACM SIGAPP Appl. Comput. Rev.*, vol. 20, no. 1, pp. 5–14, Apr. 2020, doi: 10.1145/3392350.3392351.
[93] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (Keynote paper)," in *Proc. IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 18–33, doi: 10.1109/csmr-wcre.2014.6747168.
[94] V. Bauer, T. Völke, and S. Eder, "Comparing TF-IDF and LSI as ir technique in an approach for detecting semantic re-implementations in source code," Technical University Munchen, Munich, Germany, Tech. Rep., 2015. [Online]. Available: https://mediatum.ub.tum.de/attfile/1281127/incoming/2015-Nov/626613.pdf and https://scholar.google.com/scholar?as_q=COMPARING+TF-IDF+AND+LSI+AS+IR+TECHNIQUE+IN+AN+APPROACH+FOR+DETECTING+SEMANTIC+RE-IMPLEMENTATIONS+IN+SOURCE+CODE&as_occt=title&hl=en&as_sdt=0%2C31
[95] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. 30th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, 2008, pp. 321–330, doi: 10.1145/1368088.1368132.
[96] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: Memory comparison-based clone detector," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, 2011, pp. 301–310, doi: 10.1145/1985793.1985835.

[97] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. 22Nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA, 2014, pp. 389–400, doi: 10.1145/2635868.2635900.

[98] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1157–1177, Jan. 2017.

[99] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: Detecting similarly behaving software," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA, 2016, pp. 702–714, doi: 10.1145/2950290.2950321.

[100] T. Kamiya, "Agec: An execution-semantic clone detection tool," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 227–229.

[101] R. Tekchandani, R. Bhatia, and M. Singh, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis," *J. Supercomput.*, vol. 74, no. 9, pp. 4199–4226, 2018, doi: 10.1007/s11227-016-1832-6.

[102] T. Wang, K. Wang, X. Su, and P. Ma, "Detection of semantically similar code," *Frontiers Comput. Sci.*, vol. 8, no. 6, pp. 996–1011, 2014.

[103] J. Richenhagen, B. Rumpe, A. Schloßer, C. Schulze, K. Thissen, and M. von Wenckstern, "Test-driven semantical similarity analysis for software product line extraction," in *Proc. 20th Int. Syst. Softw. Product Line Conf. (SPLC)*, 2016, pp. 174–183.

[104] P. Schugerl, J. Rilling, and P. Charland, "Reasoning about global clones: Scalable semantic clone detection," in *Proc. IEEE 35th Annu. Comput. Softw. Appl. Conf.*, Jul. 2011, pp. 486–491.

[105] R. Elva. (2013). *Detecting Semantic Method Clones in Java Code Using Method IOE-Behavior*. [Online]. Available: https://stars.library.ucf.edu/etd/2620

[106] J. Andersen, P. Bahr, F. Henglein, and T. Hvitved, "Domain-specific languages for enterprise systems," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, T. Margaria and B. Steffen, Eds. Berlin, Germany: Springer, 2014, pp. 73–95.

[107] M. Ozkaya and F. Erata, "Understanding Practitioners' challenges on software modeling: A survey," *J. Comput. Lang.*, vol. 58, Jun. 2020, Art. no. 100963. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S259011842030023X

[108] M. Voelter, J. Warmer, and B. Kolb, "Projecting a modular future," *IEEE Softw.*, vol. 32, no. 5, pp. 46–52, Sep. 2015.

[109] P. Tonella, "Evolutionary testing of classes," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 119–128, Jul. 2004, doi: 10.1145/1013886.1007528.

[110] S. Chiba, "Javassist—A reflection-based programming wizard for Java," in *Proc. ACM OOPSLA Workshop Reflective Program. C++ Java*, Oct. 1998, p. 21.

[111] J. Wu, L. Huang, and D. Wang, "ASM-based model of dynamic service update in OSGI," *SIGSOFT Softw. Eng. Notes*, vol. 33, no. 2, pp. 8:1–8:8, Mar. 2008, doi: 10.1145/1350802.1350815.

[112] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 317–328, doi: 10.1145/3238147.3238213.

[113] D. Lau. (2018). *An Abstract Syntax Tree Generator From Java Bytecode*. Accessed: Jul. 16, 2020. [Online]. Available: https://github.com/davidlau325/BytecodeASTGenerator

[114] G. M. K. Selim, K. C. Foo, and Y. Zou, "Enhancing source-based clone detection using intermediate representation," in *Proc. 17th Work. Conf. Reverse Eng.*, Oct. 2010, pp. 227–236.

[115] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 466–480, Jun. 2005.

[116] T. Muske, R. Talluri, and A. Serebrenik, "Repositioning of static analysis alarms," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2018, pp. 187–197, doi: 10.1145/3213846.3213850.

[117] A. A. Elkhail, J. Svacina, and T. Cerny, "Intelligent token-based code clone detection system for large scale source code," in *Proc. Conf. Res. Adapt. Convergent Syst.*, New York, NY, USA, Sep. 2019, doi: 10.1145/3338840.3355654.

[118] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Softw. Eng.*, vol. 16, no. 3, pp. 325–364, Jun. 2011, doi: 10.1007/s10664-010-9143-7.

[119] H. Kagdi, "Improving change prediction with fine-grained source code mining," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2007, pp. 559–562, doi: 10.1145/1321631.1321742.

[120] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-large-scale software repository and source-code mining," *ACM Trans. Softw. Eng. Methodology*, vol. 25, no. 1, pp. 1–34, Dec. 2015, doi: 10.1145/2803171.

[121] K. S. Kumar and D. Malathi, "A novel method to find time complexity of an algorithm by using control flow graph," in *Proc. Int. Conf. Tech. Advancements Comput. Commun. (ICTACC)*, Apr. 2017, pp. 66–68.

[122] M. M. Syaikhuddin, C. Anam, A. R. Rinaldi, and M. E. B. Conoras, "Conventional software testing using white box method," *Kinetik, Game Technol., Inf. Syst., Comput. Netw., Comput., Electron., Control*, vol. 3, no. 1, pp. 65–72, 2018. [Online]. Available: http://kinetik.umm.ac.id/index.php/kinetik/article/view/231

[123] J. Smits and E. Visser, "FlowSpec: Declarative dataflow analysis specification," in *Proc. 10th ACM SIGPLAN Int. Conf. Softw. Lang. Eng. (SLE)*, 2017, p. 221, doi: 10.1145/3136014.3136029.

[124] H. Shahriar, A. B. M. K. Islam Riad, M. A. I. Talukder, H. Zhang, and Z. Li, "Automatic security bug detection with findsecuritybugs plugin," Kennesaw State Univ., Kennesaw, GA, USA, Tech. Rep., Oct. 2019. [Online]. Available: https://digitalcommons.kennesaw.edu/cgi/viewcontent.cgi?article=1103&context=ccerp

[125] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Softw.*, vol. 35, no. 3, pp. 56–62, May 2018.

[126] L. Kumar, S. K. Rath, and A. Sureka, "Using source code metrics and multivariate adaptive regression splines to predict maintainability of service oriented software," in *Proc. IEEE 18th Int. Symp. High Assurance Syst. Eng. (HASE)*, 2017, pp. 88–95.

[127] K. Cemus, T. Cerny, and M. J. Donahoo, "Automated business rules transformation into a persistence layer," *Procedia Comput. Sci.*, vol. 62, pp. 312–318, Jan. 2015.

[128] A. Walker, M. Coffey, P. Tisnovsky, and T. Cerny, "On limitations of modern static analysis tools," in *Information Science and Applications*. Cham, Switzerland: Springer, Dec. 2019, pp. 577–586.

[129] R. Hat. (2019). *Fabric8-Analytics*. Accessed: Jul. 16, 2020. [Online]. Available: http://fabric8.io/faq/

[130] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proc. 40th Int. Conf. Softw. Eng., Companion*, May 2018, pp. 323–324, doi: 10.1145/3183440.3194991.

[131] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies," *Queue*, vol. 16, no. 4, pp. 10:44–10:65, Aug. 2018, doi: 10.1145/3277539.3277541.

[132] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, Dec. 2017, pp. 14–27, doi: 10.1145/3135974.3135977.

[133] A. Ibrahim, S. Bozhinoski, and A. Pretschner, "Attack graph generation for microservice architecture," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2019, p. 1235, doi: 10.1145/3297280.3297401.

[134] N. Alshuqayran, N. Ali, and R. Evans, "Towards micro service architecture recovery: An empirical study," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2018, pp. 4709–4747.

[135] F. Rademacher, S. Sachweh, and A. Zündorf, "A modeling method for systematic architecture reconstruction of microservice-based software systems," in *Enterprise, Business-Process and Information Systems Modeling*, S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic, Eds. Cham, Switzerland: Springer, 2020, pp. 311–326.

[136] Z. Jian and L. Chen, "A defense method against docker escape attack," in *Proc. Int. Conf. Cryptogr., Secur. Privacy (ICCSP)*, 2017, pp. 142–146, doi: 10.1145/3058060.3058085.

[137] B. T. Linetskyi Artem, "Eliminating privilage escalation to root in containers running on kubernetes," *Sci. Practical Cyber Secur. J.*, vol. 4, no. 1, pp. 87–92, Mar. 2020. [Online]. Available: https://journal.scsa.ge/papers/eliminating-privilage-escalation-to-root-in-containers-running-on-kubernetes/ and https://journal.scsa.ge/issue/march-2020/

[138] N. Qamar, J. Faber, Y. Ledru, and Z. Liu, "Automated reviewing of healthcare security policies," in *Foundations of Health Information Engineering and Systems*, J. Weber and I. Perseil, Eds. Berlin, Germany: Springer, 2013, pp. 176–193.

[139] D. Gordon. (2019). *A Security Framework for Python Applications*. Accessed: Jul. 16, 2020. [Online]. Available: https://github.com/YosaiProject/yosai

[140] E. W. Dijkstra, *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Oct. 1976. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/013215871X
[141] L. DeMichiel. (2017). *JSR 318: Enterprise Javabeans 3.1/Interceptors 1.2*. Accessed: Jul. 16, 2020. [Online]. Available: http://jcp.org/en/jsr/detail?id=318
[142] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with ASPECTJ," *SIGPLAN Not.*, vol. 37, no. 11, pp. 174–190, Nov. 2002, doi: 10.1145/583854.582437.
[143] M. Nishizawa, S. Chiba, and M. Tatsubori, "Remote pointcut: A language construct for distributed AOP," in *Proc. 3rd Int. Conf. Aspect-oriented Softw. Develop. (AOSD)*, 2004, pp. 7–15, doi: 10.1145/976270.976274.
[144] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli, "JAC: An aspect-based distributed dynamic framework," *Softw., Pract. Exper.*, vol. 34, no. 12, pp. 1119–1148, Oct. 2004. [Online]. Available: https://hal.inria.fr/inria-00000042
[145] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée, "Explicitly distributed aop using awed," in *Proc. 5th Int. Conf. Aspect-oriented Softw. Develop. (AOSD)*, New York, NY, USA, 2006, pp. 51–62, doi: 10.1145/1119655.1119665.
[146] E. Tanter and R. Toledo, "A versatile kernel for distributed aop," in *Distributed Applications and Interoperable Systems*, F. Eliassen and A. Montresor, Eds. Berlin, Germany: Springer, 2006, pp. 316–331.
[147] B. Lagaisse and W. Joosen, "True and transparent distributed composition of aspect-components," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware (Middleware)*. New York, NY, USA: Springer-Verlag, 2006, pp. 42–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=1515984.1515989
[148] R. Mondéjar, P. García-López, C. Pairot, and L. Pamies-Juarez, "Damon: A distributed AOP middleware for large-scale scenarios," *Inf. Softw. Technol.*, vol. 54, no. 3, pp. 317–330, Mar. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584911002138
[149] A. O. Al-Zaghameem, "Supporting distributed aspects by extending object teams model into distributed environments," Technical University of Berlin, Berlin, Germany, Tech. Rep., 2011. [Online]. Available: http://cogprints.org/7373/1/ICICS.paper/9.pdf
[150] F. Yang, H. Masuhara, T. Aotani, F. Nielson, and H. R. Nielson, "Aspectke: Security aspects with program analysis for distributed systems," in *Proc. 9th Workshop Aspects, Compon., Patterns Infrastruct. Softw. (ACP4IS)*, vol. 10, 2010, pp. 27–31.
[151] N. Tabareau, "A theory of distributed aspects," in *Proc. 9th Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, New York, NY, USA, 2010, pp. 133–144, doi: 10.1145/1739230.1739246.
[152] W. B., S. J., and D. J., "DCAM: A distributed aspectual middleware," *Int. J. Digit. Content Technol. Appl.*, vol. 4, no. 8, pp. 65–78, Nov. 2010. [Online]. Available: http://www.aicit.org/jdcta/ppl/JDCTA0408_07.pdf
[153] R. Mondejar, P. Garcia, C. Pairot, P. Urso, and P. Molli, "Designing a distributed AOP runtime composition model," in *Proc. ACM Symp. Appl. Comput. (SAC)*, New York, NY, USA, 2009, pp. 539–540, doi: 10.1145/1529282.1529395.
[154] R. Mondéjar, P. García, C. Pairot, and A. F. G. Skarmeta, "Building a distributed AOP middleware for large scale systems," in *Proc. Workshop Next Gener. Aspect Oriented Middleware (NAOMI)*, New York, NY, USA, 2008, pp. 17–22, doi: 10.1145/1408620.1408624.
[155] N. C. Narendra, K. Ponnalagu, J. Krishnamurthy, and R. Ramkumar, "Run-time adaptation of non-functional properties of composite Web services using aspect-oriented programming," in *Proc. Int. Conf. Service-Oriented Comput.* Cham, Switzerland: Springer, 2007, pp. 546–557.
[156] B. Verheecke, W. Vanderperren, and V. Jonckers, "Unraveiling crosscutting concerns in Web services middleware," *IEEE Softw.*, vol. 23, no. 1, pp. 42–50, Jan. 2006.
[157] B. Surajbali, G. Coulson, P. Greenwood, and P. Grace, "Augmenting reflective middleware with an aspect orientation support layer," in *Proc. 6th Int. Workshop Adapt. Reflective Middleware Held ACM/IFIP/USENIX Int. Middleware Conf. (ARM)*, 2007, pp. 1–6, doi: 10.1145/1376780.1376781.
[158] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," 2017, *arXiv:1709.09029*. [Online]. Available: http://arxiv.org/abs/1709.09029
[159] H. Zhang, S. Jiang, and R. Jin, "An improved static program slicing algorithm using stack trace," in *Proc. IEEE 2nd Int. Conf. Softw. Eng. Service Sci.*, Jul. 2011, pp. 563–567, doi: 10.1109/ICSESS.2011.5982378.
[160] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," in *IEEE/ACM Int. Conf. Tech. Debt (TechDebt)*, 2020, pp. 1–5.
[161] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2016, pp. 57–66, doi: 10.1109/ICDCS.2016.11.
[162] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proc. 16th Workshop Hot Topics Oper. Syst.*, New York, NY, USA, May 2017, pp. 30–36, doi: 10.1145/3102980.3102986.
[163] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger, "Finding race conditions in erlang with quickcheck and pulse," *ACM Sigplan Notices*, vol. 44, no. 9, pp. 149–160, 2009.
[164] C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Santa Clara, CA, USA: USENIX Association, Mar. 2016, pp. 291–309. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/scott
[165] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surveys*, vol. 43, no. 2, pp. 1–29, Jan. 2011.
[166] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, "Constrained interaction testing: A systematic literature study," *IEEE Access*, vol. 5, pp. 25706–25730, 2017.
[167] B. S. Ahmed, A. Gargantini, K. Z. Zamli, C. Yilmaz, M. Bures, and M. Szeles, "Code-aware combinatorial interaction testing," *IET Softw.*, vol. 13, no. 6, pp. 600–609, Dec. 2019.
[168] M. Bures and B. S. Ahmed, "On the effectiveness of combinatorial interaction testing: A case study," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2017, pp. 69–76.
[169] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, J. J. Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013.
[170] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2014, pp. 79–90.
[171] S. Wang and J. Offutt, "Comparison of unit-level automated test generation tools," in *Proc. Int. Conf. Softw. Test., Verification, Validation Workshops*, 2009, pp. 210–219.
[172] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, Mar. 2012.
[173] G. Márquez and H. Astudillo, "Identifying availability tactics to support security architectural design of microservice-based systems," in *Proc. 13th Eur. Conf. Softw. Archit. (ECSA)*, New York, NY, USA: Association for Computing Machinery, vol. 2, 2019, pp. 123–129, doi: 10.1145/3344948.3344996.
[174] T. Cerny. (2019). *Microservice Testbed for Texas Teacher Examination*. Accessed: Jul. 16, 2020. [Online]. Available: https://bitbucket.org/advseproject/documentation/
[175] M. I. Rahman, S. Panichella, and D. Taibi, "A curated dataset of microservices-based systems," in *Proc. Joint Inforte Summer School Softw. Maintenance Evol. (CEUR-WS)*, vol. 2520, 2019, pp. 1–9.
[176] L. B. Othmane, P. Angin, H. Weffers, and B. Bhargava, "Extending the agile development process to develop acceptably secure software," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 6, pp. 497–509, Nov. 2014.
[177] H. Oueslati, M. M. Rahman, and L. B. Othmane, "Literature review of the challenges of developing secure software using the agile approach," in *Proc. 10th Int. Conf. Availability, Rel. Secur.*, Aug. 2015, pp. 540–547.
[178] Veracode. (2019). *Veracode—Penetration Testing Tool*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.veracode.com/security/penetration-testing
[179] Tenable. (2019). *Tenable—Penetration Testing Tool*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.tenable.com/products/nessus/nessus-professional
[180] PortSwigger. (2019). *Portswigger—Web Vulnerability Scanner*. Accessed: Jul. 16, 2020. [Online]. Available: https://portswigger.net
[181] Checkmarx. (2019). *Checkmarx Software Exposure Platform*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.checkmarx.com
[182] OWASP Foundation. (2020). *The Open Web Application Security Project*. [Online]. Available: https://owasp.org
[183] A. Majd, M. Vahidi-Asl, A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, "Code4Bench: A multidimensional benchmark of codeforces data for different program analysis techniques," *J. Comput. Lang.*, vol. 53, pp. 38–52, Aug. 2019.

[184] D. Taibi. (2020). *A Curated List of Open Source Projects Developed With a Microservices Architectural Style*. Accessed: Jul. 16, 2020. [Online]. Available: https://github.com/davidetaibi/Microservices_Project_List

TOMAS CERNY was an Assistant Professor of computer science with the Czech Technical University, FEE, Prague, Czech Republic, from 2009 to 2017. In 2017, he held a postdoctoral position at Baylor University, TX, USA. In 2017, he continued as an Assistant Professor with the concentration on software engineering. He is currently a Professor. He has authored nearly 100 publications mostly related to code analysis and aspect oriented programming.

Dr. Cerny served over ten years as the lead developer for the International Collegiate Programming Contest Management System. He received the Outstanding Service Award ACM SIGAPP 2018 and 2015 or the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. In the past few years, he chaired multiple conferences, including ACM SAC, ACM RACS, or ICITCS. He led special issues and track on code analysis and enterprise applications.

JAN SVACINA received the bachelor's degree from the Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University. He is currently pursuing degree in computer science with Baylor University. His research interests include software engineering, microservice security, and code analysis. In 2020, he received the Outstanding Graduate Student Award from Baylor University.

DIPTA DAS received the bachelor's degree from the Department of Computer Science and Engineering, Chittagong University of Engineering and Technology, Chittagong, Bangladesh. He is currently pursuing degree in computer science with Baylor University. His research interests include software engineering, microservice security, and code analysis.

VINCENT BUSHONG received the bachelor's degree from Evangel University, Springfield, MO, USA. He is currently pursuing degree in computer science with Baylor University. His research interests include microservices and code analysis.

MIROSLAV BURES was a Visiting Researcher with Abertay University, U.K., in 2018. He leads the Software Testing IntelLigent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University, Prague. In 2010, he was appointed at the Czech Technical University in Prague, where he is currently an Associate Professor of computer science. He leads several projects in the field of test automation for software and the Internet of Things systems, covering the topics of automated generation of test scenarios as well as automated execution of the tests. He has authored over 60 research publications and wrote two books on software testing and web technologies. His research interests include quality assurance and reliability methods, model-based testing, path-based testing, combinatorial interaction testing and test automation for software, the Internet of Things, and mission-critical systems.

Prof. Bures is a member of the Czech Chapter of ACM, the Czech and Slovak Testing Board, and the ISTQB Academia Committee.

PAVEL TISNOVSKY received the Ph.D. degree from the Brno University of Technology, Czech Republic.

He was an Assistant Professor, from 1999 to 2005. He is currently a Principal Quality Engineer with Red Hat, Inc., with over ten years of experience. He is a programming language enthusiast and the author of many articles and series at Linux magazine ROOT.cz. He holds one software patent on testing and currently works on tools for OpenShift.io—open development services for creating, building, and testing container applications.

KAREL FRAJTAK received the master's and Ph.D. degrees from the Faculty of Electrical Engineering, Czech Technical University, Prague.

Since 2017, he has been a Lecturer and a Researcher with the STILL Group. He is currently a Lecturer and a Researcher with the Software Testing IntelLigent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University, Prague. He has over 15 years' experience in software development in the industry and is currently applying this experience in the academic world. His lectures and area of research focus on the software testing methods and test automation approaches. He has authored or coauthored publications published in peer-reviewed journals.

DONGWAN SHIN received the master's and Ph.D. degrees in computer science from the University of North Carolina at Charlotte, in 1999 and 2004, respectively. He is currently an Associate Professor with the Computer Science and Engineering Department, New Mexico Tech. His primary research interests include the areas of system and software security. In 2005, he joined the Computer Science and Engineering Department, New Mexico Tech, where he has been extensively involved in a variety of research and educational projects funded by various government agencies, national labs, and industry partners on cybersecurity. The research results have been successfully presented and published at leading conferences and journals. He has been leading efforts in different roles to organize many ACM and IEEE conferences and workshops, such as ACM SAC, SACMAT, CCS, and CloudApp. He was a recipient of the ACM SIGAPP Distinguished Service Award and the Outstanding Service Awards in the past. He has led departmental efforts in developing a successful cybersecurity program at New Mexico Tech, which resulted in New Mexico Tech as a national Center of Academic Excellence in Information Assurance Education and Research (CAE and CAE-r) designated by NSA and DHS. He is currently the point of contact of the center. He was named the Orr Endowed Chair of the Computer Science and Engineering Department while serving as the Department Chair from 2015 to 2018.

JUN HUANG (Senior Member, IEEE) received the Ph.D. degree (Hons.) from the Institute of Network Technology, Beijing University of Posts and Telecommunications, China, in 2012.

He was a Visiting Scholar with the Global Information and Telecommunication Institute, Waseda University, a Research Fellow with the Electrical and Computer Engineering Department, South Dakota School of Mines and Technology, a Visiting Scholar with the Computer Science Department, The University of Texas at Dallas, and a Guest Professor with the National Institute of Standards and Technology. He is currently a Full Professor of computer science with the Chongqing University of Posts and Telecommunications. He received the Best Paper Award from EAI Mobimedia 2019, the Outstanding Service Award from ACM RACS 2017, 2018, and 2019, the Best Paper Nomination from ACM SAC 2014, and the Best Paper Award from AsiaFI 2011. He has authored over 120 publications, including papers in prestigious journal/conferences, such as the IEEE TWireless, NetMag, ComMag, WCM, VTM, the IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS, TBC, IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, TNSM, TSUSC, IoTJ, TCC, TCCN, IWQoS, SCC, ICCCN, GLOBECOM, ICC, ACM SAC, and RACS. He is an Associate Editor of IEEE ACCESS and the *KSII Transactions on Internet and Information Systems*. He guest-edited several special issues on IEEE journals, such as the IEEE NETWORK, the *IEEE Communications Magazine*, the IEEE INTERNET OF THINGS JOURNAL, and IEEE ACCESS. He also chaired and co-chaired multiple conferences in the communications and networking areas and organized multiple workshops at major IEEE and ACM events. His current research interests include network optimization and control, device-to-device communications, and the Internet of Things.