Infrastructure From Code

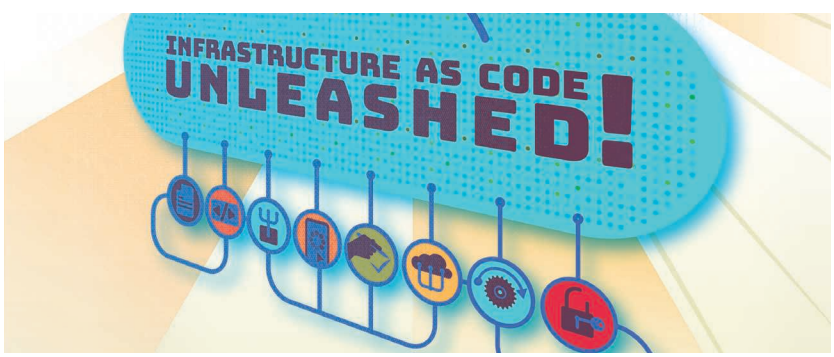
The Next Generation of Cloud Lifecycle Automation

Itzhak Aviv, **Ruti Gafni**, **and Sofia Sherman**, The Academic College of Tel Aviv-Yaffo

Berta Aviv, Human Centric Management Consulting

Asher Sterkin and Etzik Bega, BlackSwan Technologies

// We identify 14 fundamental cloud infrastructure procedures (CIPs) applicable to software development processes on the public cloud and their associated challenges. We then evaluate the capabilities of leading cloud automation technologies, such as infrastructure as code, and pinpoint their gaps in enabling the CIPs. //



Digital Object Identifier 10.1109/MS.2022.3209958
Date of current version: 23 December 2022

OVER THE LAST decade, cloud computing has become more popular among developers, creating a new breed of cloud-native applications that promise greater flexibility and efficiency while being more cost effective and fault tolerant.[1,2,3] Cloud-native application development teams must deal with various cloud infrastructure procedures (CIPs). Cloud infrastructure deployment from scratch requires specialized knowledge and skills to design, develop, and integrate cloud infrastructure, which is complex, error-prone, and time-consuming.[2,3,4,5,6] That application lifecycle, therefore, demands considerable infrastructure procedure automation to simplify development and deployment for the cloud.

Organizations' cloud computing implementation efforts focus on the "need-for-speed": faster deployment, faster release cycles, and faster recovery.[1] The latest research suggests methods to simplify cloud infrastructure management: command-line interface scripting (CLI),[1,2] low code/no-code platforms (LPC),[3,4] continuous integration and deployment (CI/CD),[5,6] and infrastructure as code (IaC),[7,8,9,10,11,12] Despite these methods, cloud deployment challenges remain,[1,3,4,5,6,8,9,11] To reveal the current state of the art, we formulated the following research questions:

RQ1. What are the CIPs that challenge the cloud-native community?
RQ2. How do leading cloud deployment automation technologies fulfill the CIPs' challenges?

We utilized the grounded theory (GT) research[13,14] approach to

expose 14 CIPs that challenge cloud-native developers. We then analyzed the leading cloud deployment automation approaches addressing CIPs by applying metaprocess analysis.[15]

Accordingly, we propose a new infrastructure from code (IfC) approach to address the existing gaps in supporting the CIPs. Whereas IaC deployment requires professional skills in cloud architecture followed by code or model abstraction, IfC automatically generates cloud infrastructure and deployment scripts from application source code, eliminating the need for a domain-specific language and error-prone manual operations.

With IfC, cloud infrastructure code and deployment scripts are generated automatically from the functional service source code itself, eliminating the need for a different language or platform for deployment automation. In IfC, the default configuration is assigned to each cloud resource in a holistic approach covering all aspects required for its deployment: basic resource creation, relevant identity access definitions, security configuration, logging, and so on. A default configuration approach is highly suitable for practical development environments. Skilled users administering more complex systems can modify the default configuration and change the parameters in the IfC configuration file according to their preferences or policy. Examples of this are changing the level of security by the parameters in the IfC configuration file during the deployment-to-production transition or selecting an SQL database instead of semistructured cloud storage.

This article offers practitioners the following insights: 1) defining the current CIP challenges for software teams, 2) introducing the novel IfC paradigm, and 3) evaluating the impact of IfC through an empirical use case.

Cloud Infrastructure Procedures (CIPs)

To answer RQ1, we used a GT[13] approach protocol[14] to define key CIPs. GT formulates new premises inductively from the data collected in the investigated field. Sixteen open-ended interviews were conducted with cloud-native system development team leaders, system architects, system engineers, and developers implementing a code-centric deployment platform (83% agreed to be interviewed). We used in vivo[13] open coding to categorize transcribed interviews and defined 76 open codes. Then, we apply axial coding to create categories from the identified codes (defined 14 CIPs), followed by selective coding to cluster the categories into groups and identify each group's main category (defined four main categories). Validation consists of the GT's consolidation: "After much fitting of words, validity is achieved when the selected word best represents the pattern. It is equally valid and grounded."[13] The data are inductively coded until data analysis reaches saturation.[14] This section presents 14 CIP categories grouped into four main categories that receive the most attention from the cloud-native community.

- Category 1: Cloud Infrastructure Skills
  - CIP1: Internal Experts
  - CIP2: External Consulting
- Category 2: Cloud Infrastructure Deployment
  - CIP3: Architecture Design
  - CIP4: Infrastructure Setup
  - CIP5: Infrastructure Security
- Category 3: Application Development and Deployment
  - CIP6: Development Environment Setup
  - CIP7: Development and Testing
  - CIP8: Deployment
  - CIP9: Migration
  - CIP10: Security
  - CIP11: Data Protection
- Category 4: Cloud Performance and Cost
  - CIP12: Performance Optimization
  - CIP13: Cost Optimization
  - CIP14: Infrastructure Portability

Category 1: Cloud Infrastructure Skills

Acquiring cloud infrastructure expertise is challenging. A successful cloud transition requires professional cloud experience, including setting up networks, security, compute, hardware, storage, and virtualization software. Participants noted the lack of internal cloud experts (CIP1) and the high cost of hiring external consultants (CIP2).

Category 2—Cloud Infrastructure Deployment

Cloud infrastructure must be deployed quickly to adapt, utilizing automation platforms. The architectural design phase (CIP3) is challenging in selecting cloud services and mapping the flow between application logic and cloud services. This step incorporates naming conventions, styling, structuring, and formatting.

Infrastructure setup (CIP4) is challenging due to the sheer complexity of resource configuration specifications, which can span hundreds or thousands of lines of code across multiple owners and files. Errors can cascade across environments and applications. Subsequently, versioning operations such as inserting, erasing, modifying

cloud resources, and revising their topology are challenging and impose on participants' responsibilities.

Infrastructure Security (CIP5) is challenging for configuring security need to configure practices addressing access privileges, data encryption, secured fact access, secret management, and secure logging, among other considerations.

Category 3—Apps Development and Deployment

Environment setup (CIP6) involves traditional development, test, staging, and production environments. Developers need to modify their code during the cloud software lifecycle to operate within these environments—a challenging task because each environment warrants distinctly fine-tuning the configuration.

CIP7 addresses architect-identified cloud resources and services, which must be learned, applied, and coded. Developers must acquire, update, and deploy code libraries. They must test locally on their workstations and in the cloud. Thus, code must often be modified by creating and managing deployment pipelines (CIP8).

Legacy applications and data stores developed for older architectures are CIP9's fundamental issues. Organizations often are expected to choose between a "lift-and-shift" of existing assets as-is, all-at-once conversion to cloud-native computing ("re-platforming"), or a hybrid strategy.

Cloud transformation affects application security and data protection (CIP10, 11). In the "shared responsibility" paradigm, the customer is responsible for data and workload security. The cloud provider offers certain advanced security features, but the customer is responsible for configuring them correctly and protecting all services and applications.

> A successful cloud transition requires professional cloud experience, including setting up networks, security, compute, hardware, storage, and virtualization software.

Category 4—Performance and Cost

Cloud migration challenges include performance optimization (CIP12) and cloud cost management (CIP13). Participants assign applications, data, and cloud resources to different vendors for cost and technical reasons, continually reassessing needs. However, vendor lock-in hinders portability (CIP14). Deployed cloud services aren't easily transferable among cloud platforms, so migration of cloud workloads is usually time-consuming and expensive.

How Existing Cloud Deployment Approaches Support The CIPs

RQ2 examines how leading cloud deployment automation approaches fulfill the CIPs' challenges. We used a metaprocess model[15] to map the adjustment of existing approaches to 14 CIPs.

The community still uses command-line interface scripting (CLIs),[1] a common, power-user-friendly alternative to graphical interfaces. However, CLIs have accessibility issues: unstructured text, lack of status and progress indicators, and inaccessible error messages.[2] CLIs are suitable for simple application and infrastructure deployments because they require highly skilled specialists[14] (CIP1–3) and do not cover cloud infrastructure and application automation (CIP3–9) or neither nonfunctional aspect, such as security, cost management, portability (CIP5, 11–14).

Low code platform (LCP) software development emphasizes visual logic representation, drag-and-drop component placement, value parameterization, and automated asset integration.[3] LCP enables rapid, continuous, test-and-learn delivery for market and internal needs.[4] LCP barriers include the inability of architectural diagrams to represent complex software, detailed application logic, and infrastructure environments; the superior productivity of common programming languages to capture more sophisticated logic;[4] a lack of precise representation standards across LCP vendors (CIP3, 7, 8, 14);[3] and minimal effect on applications security, data protection, and optimization (CIP10–13). Moving from low- to no-code simplifies specifications but greatly reduces development versatility.[4]

Deployment-centric productivity tools focus on CI/CD operation. CI automates the build process on dedicated servers to detect errors early and improve overall quality, while CD aids in achieving short cycle releases.[5]

Recent research assumes that code updates using different configuration languages cause failures, suggesting a synthesis-based method for automatically repairing environment configurations.[6,7,8] For example, tortoise configuration technology fixes infrastructure errors via the shell and then synchronizes the adjustments' upwards' in the configuration language specifications.[6] Other studies have recommended automating version-changed configuration script repair[7] and inferring working environment configurations without developer interaction.[6] The synthesis-based method focuses on creating and maintaining configuration scripts; however, it doesn't perform configuration drift automatically from the application code.

Productivity tools partially facilitate CIPs. Implementing these tools is not a simple procedure, though, as some flow sections may have become obsolete and need change (e.g., architecture changes, operating system updates, library upgrades—CIP3–5); new skills may also be required[9] (CIP1–3). Misapplication will reduce effectiveness and cause maintainability issues[5] (CIP12, 14). There are also security issues, i.e., code needs to be added to secure communications between containers, requiring security infrastructure configuration[9] (CIP10–11).

IaC facilitates the provisioning and management of cloud or virtual infrastructure (e.g., networks, virtual machines, load balancers, connection topology) with software by code,[10] rather than configuring hardware. A typical IaC platform implements the CI/CD pipeline, version control, and testing tools. It specifies environmental parameters, making cloud configuration faster to set up, customize, repeat, and debug. Code-centric platforms manage cloud infrastructure provisioning (e.g., Ansible/Terraform), server software installation, and management (e.g., Chef/Puppet).[10] Highly qualified specialists must write IaC configuration code or markup documents (CIP1, 2). Given the complexity of resource configuration specifications, which can span hundreds or thousands of lines across multiple owners and files, output errors may cascade across environments and applications.

Code-centric IaC falls short of the full cloud potential because it only handles resource allocation and overlooks cloud infrastructure optimization, rapid architecture design, and migration (CIP3, 4, 8, 9, 12, 13). It modestly supports security and data protection (CIP4, 10, 11).

To enhance the efficiency of IaC, the latest research proposes a model-driven approach that provides modeling tools supporting automatic infrastructure code generation transparent to developers, thus improving design setup, deployment, and portability. For example, the Cloud Application Modelling and Execution Language approach models and deploys hybrid infrastructure across multiple clouds. CloudMF exploits deployment configurations at design-time and runtime to reconfigure provisioned cloud services.[11] Practitioners are skeptical of the model-driven development benefits because it lacks validated tools and real-world evidence of effectiveness.[10] The SODALITE project (https://www.sodalite.eu/) is a widely studied, model-driven approach that aims to provide tools for faster application development, deployment, and execution on heterogeneous infrastructures. SODALITE provides a user interface to design an abstract deployment model and capture application runtime information, followed by infrastructure generation and optimization.

Ultimately, the model-driven approach still requires professional knowledge of cloud infrastructure topology, a deep understanding of security and privacy issues, and optimization experience. Its modeling interface is mostly based on a separate deployment system. Therefore, it is still limited in simplifying the cloud infrastructure configuration process.

Infrastructure From Code (IfC)

Following the identified research gaps in existing methods, we consider a novel approach called *IfC*. IfC interprets functional programming in a common language like Python, then automatically determines cloud configuration settings and generates implementation scripts. This automation layer provides numerous benefits, including measurable, order-of-magnitude improvements in development productivity and deployment speed. Further cloud management benefits can be achieved with IfC as the starting point. For example, IfC adoption assures that the specification instructions are bug-free. Security policies, including least-privilege access, can be enforced automatically by a cloud configuration compiler at a resource and role-specific level. Implementing IfC, computer-generated configuration instructions can close the loop across application deployment, performance measurement (observability), and auto-optimization at the environment and application levels. Figure 1 provides the IfC reference architecture.

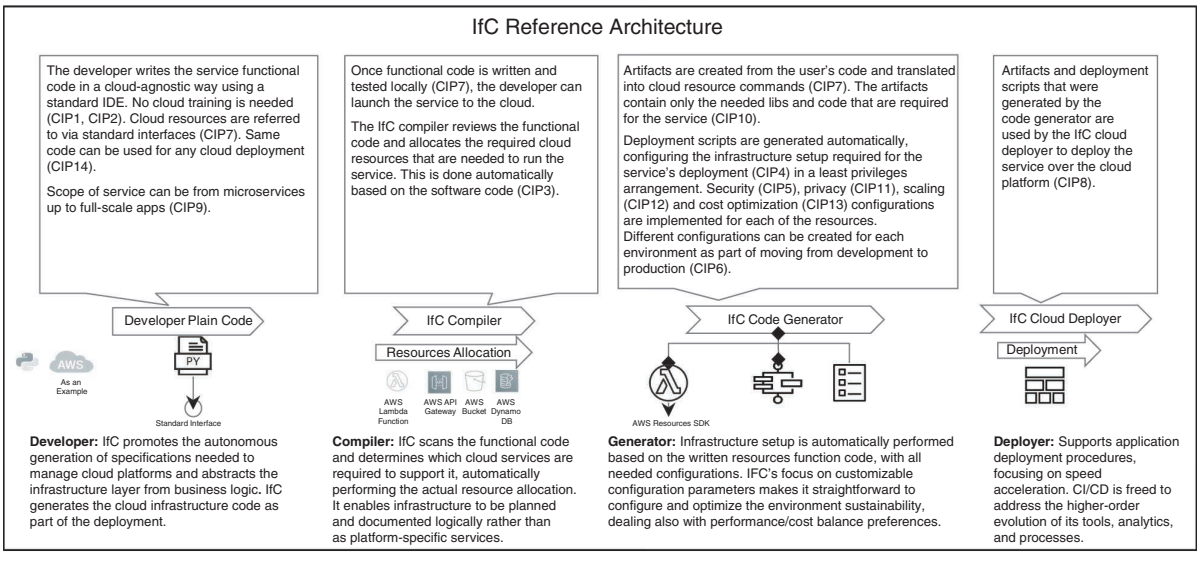


FIGURE 1. IfC reference architecture and case study results.

We conducted an empirical experiment to determine IfC's actual contribution. First, we evaluated IaC solutions presented in the Gartner research group review (Gartner IDG00720736/2021). Then we analyzed IaC startup solutions (March 2021) to determine how well they address the 14 CIP challenges identified by current research. We discovered that BST's Labs (https://blackswantechnologies.ai) cloud automation platform, which was in development, matched IfC's requirements superiorly to those of other platforms. We partnered with BST to adapt the empirical IfC evaluation through the case study. From November 2021 to January 2022, we evaluated how well the enhanced version of the Cloud Infrastructure Automation Operational System (CAIOS) platform based on the IfC principles met the requirements of the 14 CIPs through an empirical case study. We implemented the platform to support 70 undergraduate information systems students with basic Python programming skills but no prior knowledge to develop a cloud-native application. Using students as participants validly simplifies the reality needed in laboratory contexts.[12] Since they have no prior knowledge of cloud infrastructure, students can be seen as potential beneficiaries of IfC and therefore serve as a suitable audience for initial validation. The students were divided into 24 groups and worked on their course project for developing cloud-native applications using Amazon Web Services (AWS).

The students were given a rudimentary explanation of using CAIOS before receiving their course assignments. In a new bank account provisioning simulation, the assignment was to build a name-screening service for the bank to validate the customer's identity to reduce the risk of identity theft, money laundering, financial fraud, and the financing of criminal organizations. At the outset, the students were required to manipulate data from the sanctions list database, then to build a function that checks if a new client is in this database. The implementation used a REST application programming interface (API) protocol while securing the entire process using encryption and authentication.

Using CAIOS, the students created functional code within a Developer environment. The Compiler reviews the functional code and allocates cloud resources. Next, the Generator translates students' Python code to native cloud resources, such as AWS Lambda functions, S3 bucket, API gateway, and Cognito authentication. Keys seamlessly encrypted the data without the students needing to manage this. Finally, the Deployer creates artifacts, deploys scripts, and deploys student applications over the cloud. CAIOS is the only tool students use to create applications and full-stack cloud deployments without coding or scripting infrastructure operations.

In the three and a half months during the course project, the total time spent by students on functional code development was 2,856 h, while cloud-operational code (launched 109 cloud services) required only 11 h. As shown in Figure 1, we demonstrated that the utilization of IfC saved thousands of lines of cloud-operational code that would have been created using a traditional development approach.

Discussion

The research reveals 14 CIPs that represent the life cycle management of cloud infrastructure by application developers and DevOps teams. While classical approaches focus on DevOps teams, IfC provides cloud automation also for application developers. As presented in Figure 1, during the IfC case study, the application developer created only one sentence of code to allocate cloud storage without any previous knowledge of the cloud.

IfC eliminates the need to write and maintain complex configuration specifications and learn, for example, IaC languages. Instead, infrastructure and deployment scripts are automatically generated from application source code. Each cloud resource is assigned a default configuration that encompasses all aspects required for deployment, including basic resource creation, relevant identity and access roles, security configuration, logging, and so on. Skilled users involved with complex projects can alter the default configuration based on preferences or organizational policies.

The students' experiment results demonstrate IfC's value in respecting enhanced developer productivity and deployment speed. While the capabilities of existing cloud-automation approaches only partially satisfy CIPs, the proposed IfC has substantial advantages in meeting all CIPs' requirements and providing practitioners with immediate benefits. Existing methods differ from IfC because they do not suggest performing configuration drift automatically from the application code, without any requirement for cloud infrastructure skills and experience, as IfC does. The gaps highlight the IfC potential for advances over IaC to meet development teams' needs better.

Implementing the IfC platform for a small student project is a limitation of the current research. Future research should adapt IfC to large and medium-sized real-world use cases. In our vision, vendors would deliver IfC as a plug-in for most

majority of standard development environments. To provide a complete solution for complex systems and multicloud environments, IfC future research should also focus on its evolution, considering the hybrid approach of model-driven IaC and configuration-based IfC.

References

1. K. Albusays, S. Ludi, and M. Huenerfauth, "Interviews and observation of blind software developers at work to understand code navigation challenges," in *Proc. 19th Int. ACM SIGACCESS Conf. Comput. Accessibility*, 2017, pp. 91–100, doi: 10.1145/3132525.3132550.
2. H. Sampath, A. Merrick, and A. Macvean, "Accessibility of command line interfaces," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2021, pp. 1–10, doi: 10.1145/3411764.3445544.
3. Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, "Characteristics and challenges of low-code development: The practitioners' perspective," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2021, pp. 1–11, doi: 10.1145/3475716.3475782.
4. R. Sanchis, Ó. García-Perales, F. Fraile, and R. Poler, "Low-code as enabler of digital transformation in manufacturing industry," *Appl. Sci.*, vol. 10, no. 1, p. 12, Dec. 2019, doi: 10.3390/app10010012.
5. F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Softw. Eng.*, vol. 25, no. 2, pp. 1095–1135, Mar. 2020, doi: 10.1007/s10664-019-09785-8.
6. A. Weiss, A. Guha, and Y. Brun, "Tortoise: Interactive system configuration repair," in *Proc. IEEE/ACM*

ABOUT THE AUTHORS

ITZHAK AVIV is a faculty member at The Academic College of Tel-Aviv–Yaffo, Tel Aviv 6818543, Israel. His current research interests include open architecture, cloud computing, the Internet of Things, blockchain, and cybersecurity. Aviv received his Ph.D. in information systems from Haifa University. Contact him at itzhakav@mta.ac.il.

RUTI GAFNI is the Dean of the Information Systems School at The Academic College of Tel-Aviv–Yaffo, Tel Aviv 6818543, Israel. Her research interests include cybersecurity and new technologies adoption. Gafni received her Ph.D. in information systems from Bar-Ilan University. Contact her at rutigafn@mta.ac.il.

SOFIA SHERMAN is a faculty member at The Academic College of Tel-Aviv–Yaffo, Tel Aviv 6818543, Israel. Her current research interests include requirements engineering and system design. Sherman received her Ph.D. in information systems from Haifa University. Contact her at sofias@mta.ac.il.

BERTA AVIV is a people analytics consultant in Human Centric Management Consulting, Tel Aviv 6818543, Israel. Her research interests include cognition and behavioral science. Aviv received her B.A. in organizational and educational consulting. Contact her at berta500@yahoo.com.

ASHER STERKIN is the senior vice president of engineering at BlackSwan Technologies, Tel Aviv 6492105, Israel. His current research interests include cloud computing and software developments productivity tools. Sterkin received his M.Sc. in economic cybernetics from the Moscow Institute of Management. Contact him at asher@blackswan-technologies.com.

ETZIK BEGA is a Director at BlackSwan Technologies, Tel Aviv 6492105, Israel. His current research interests include cloud computing and software developments productivity tools. Bega received his M.B.A. from Ben Gurion University. Contact him at etzik@blackswan-technologies.com.

32nd Int. Conf. Automat. Softw. Eng. (ASE), 2017, pp. 625–636, doi: 10.1109/ASE.2017.8115673.
7. C. Macho et al., "Automatically repairing dependency-related build breakage," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2018, pp. 106–117, doi: 10.1109/SANER.2018.8330201.
8. E. Horton and C. Parnin, "DockerizeMe: Automatic inference of environment dependencies for python code snippets," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 328–338, doi: 10.1109/ICSE.2019.00047.
9. M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 197–207, doi: 10.1145/3106237.3106270.
10. J. Sandobalin, E. Insfran, and S. Abrahao, "On the effectiveness of tools to support Infrastructure as code: Model-driven versus code-centric," *IEEE Access*, vol. 8, pp. 17,734–17,761, Jan. 2020, doi: 10.1109/ACCESS.2020.2966597.
11. A. Bergmayr et al., "A systematic review of cloud modeling languages," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–38, Jan. 2019, doi: 10.1145/3150227.
12. D. Falessi et al., "Empirical software engineering experts on the use of students and professionals in experiments," *Empirical Softw. Eng.*, vol. 23, no. 1, pp. 452–489, Feb. 2018, doi: 10.1007/s10664-017-9523-3.
13. J. Corbin and A. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Newbury Park, CA, USA: Sage, 2014.
14. I. Aviv, I. Hadar, and M. Levy, "Knowledge management infrastructure framework for enhancing knowledge-intensive business processes," *Sustainability*, vol. 13, no. 20, p. 11,387, Oct. 2021, doi: 10.3390/su132011387.
15. M. Levy, I. Hadar, and I. Aviv, "A requirements engineering methodology for knowledge management solutions: Integrating technical and social aspects," *Requirements Eng.*, vol. 24, no. 4, pp. 503–521, Dec. 2019, doi: 10.1007/s00766-018-0298-x.

Digital Object Identifier 10.1109/MS.2022.3203246