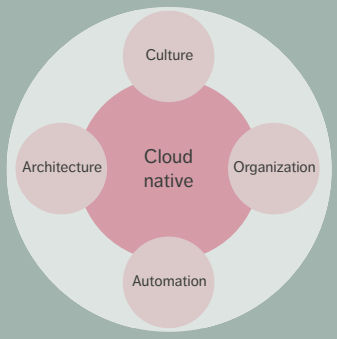
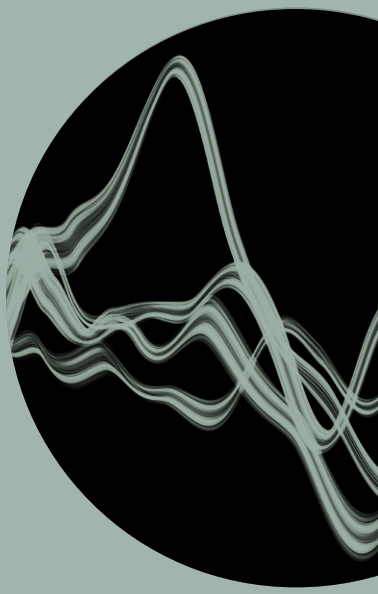
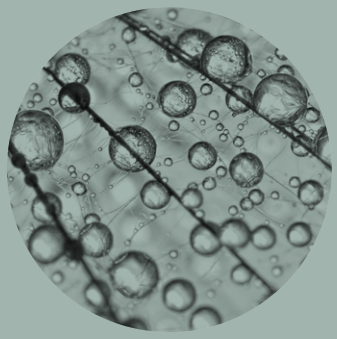
CHARTING THE FUTURE OF INNOVATION | #05·2019

ERICSSON TECHNOLOGY Review





CLOUD-NATIVE APPLICATION DESIGN



ERICSSON

Cloud-native application design

IN THE TELECOM DOMAIN

Cloud-native application design is set to become common practice in the telecom industry in the next few years due to the major efficiency gains that it can provide, particularly in terms of speeding up software upgrades and releases.

HENRIK SAAVEDRA PERSSON, HOSSEIN KASSAEI

The cloud-native paradigm is driving the transformation of virtual network functions into cloud-native applications (CNAs) that can be commercialized and offered according to either as-a-service (aaS) or as-a-product (aaP) models. In either case, the goal is to provide a seamless and secure deployment, monitoring and operations experience by applying a very high degree of automation.

To ease the transition to the cloud-native approach, Ericsson has created an application development framework that provides a set of architecture principles, design rules and best practices that guide the fundamental design decisions for all of our CNAs. Our framework leverages web-scale technology from the Cloud Native Computing Foundation (CNCF) and other open-source projects while taking into consideration the particular challenges of production-grade telecom applications.

The CNCF is an open-source software foundation whose stated purpose is to make cloud-native computing 'universal and sustainable.' It fosters collaboration between the industry's top developers, end users, and vendors, serving as the vendor-neutral home for many of the fastest-growing projects on GitHub, including Kubernetes, Prometheus and Envoy. CNCF technology has played an important role in our efforts to develop and refine our approach to CNA design.

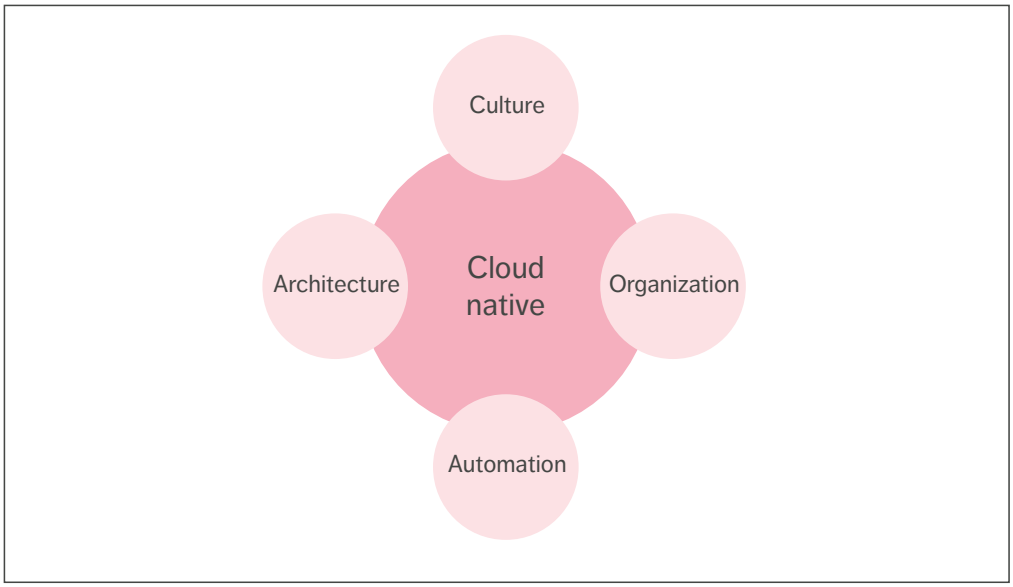


Figure 1 The four pillars of the cloud-native paradigm

Figure 1 illustrates the four pillars of the cloud-native paradigm. Our framework addresses three of them: automation, architecture and culture. Automation is an integral part of the framework, which takes a CI/CD (Continuous Integration, Continuous Delivery) approach to application development and delivery. Architecturally, the framework provides the software assets/components that enable applications to fulfill key design principles [1]. Culturally, it promotes collaboration with the open-source community, as using and contributing to the relevant open-source software projects (typically within CNCF) is at the heart of our implementation strategy.

Our application development framework

Our framework establishes a set of principles for telecom applications based on microservices, containers and state-optimized design. It provides a set of best practices, design rules and guidelines on

Terms and abbreviations

AAP – As-a-Product | **AAS** – As-a-Service | **ACID** – Atomicity, Consistency, Isolation, and Durability | **CAP** – Consistency, Availability and Partition Tolerance | **CAT** – Configuration Assessment Tool | **CI/CD&D** – Continuous Integration, Continuous Delivery and Deployment | **CIS** – The Center for Internet Security | **CNA** – Cloud-native Application | **CNCF** – Cloud Native Computing Foundation | **DR** – Design Rule | **ETSI** – European Telecommunications Standards Institute | **MSA** – Microservice Architecture | **NIST** – National Institute of Standards and Technology | **UI** – User Interface

how to build CNAs based on microservice architecture (MSA), as well as guidance on how to deploy, monitor and operate them based on DevOps principles.

With the support of our framework, it is possible to build telecom applications that use CNCF technology through a highly modular architecture and clear separation of concerns. The framework helps us drive alignment across all Ericsson CNAs, ensuring that we address key concerns in a common, generic way. The consistent life-cycle management, operation and maintenance that result from this approach enhance the customer experience. *Figure 2* provides a high-level picture of what the framework offers.

Designing cloud-native applications

Ericsson CNAs are built as a set of loosely coupled (micro)services with well-defined, bounded contexts and individual life cycles. Each microservice is packaged and delivered as one or more containers, independent from other microservices, and provides well-defined and version-controlled application programming interfaces exposed over the network.

To achieve full portability across various infrastructures, CNAs rely on Kubernetes as the choice of container orchestration platform and can be deployed on any certified Kubernetes distribution [2] with a minimum version adhering to the company's security and stability requirements.

All Ericsson CNAs are fully verified on Ericsson Kubernetes distribution. Our CNAs rely on Kubernetes for the automatic placement, auto-scaling, upgrade and auto-healing of individual services. On top of making use of Kubernetes, we also contribute features back to Kubernetes that make it a better fit for telco-grade deployments. IPv6 is just one example of an important area within the telecom domain that has not yet received enough attention within the community.

Observability, security and persistence

Observability is a prerequisite for seamless CNA monitoring and operations. The CNCF landscape [3] includes several very good candidates to help collect,

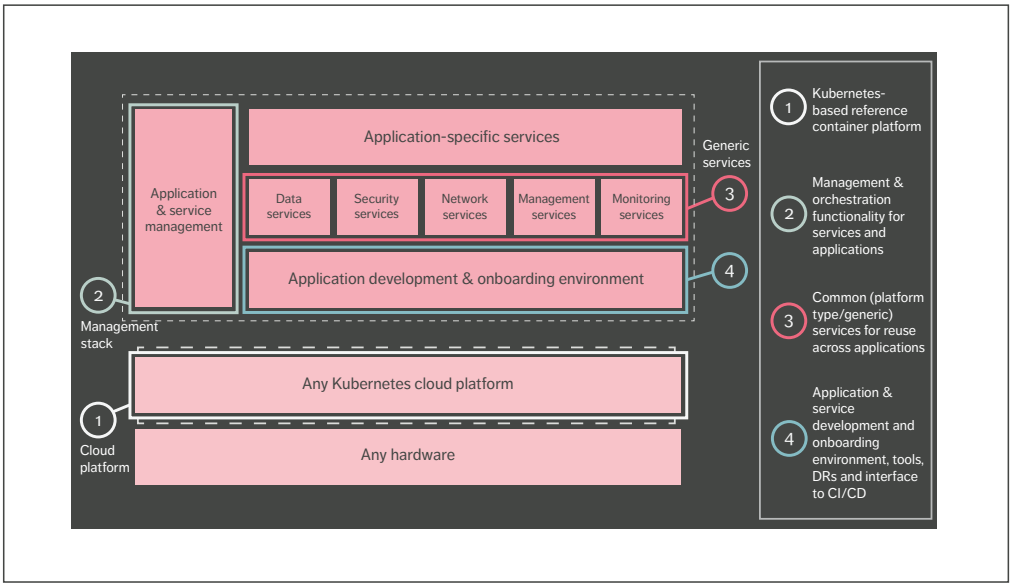


Figure 2 Key components of Ericsson's application development framework

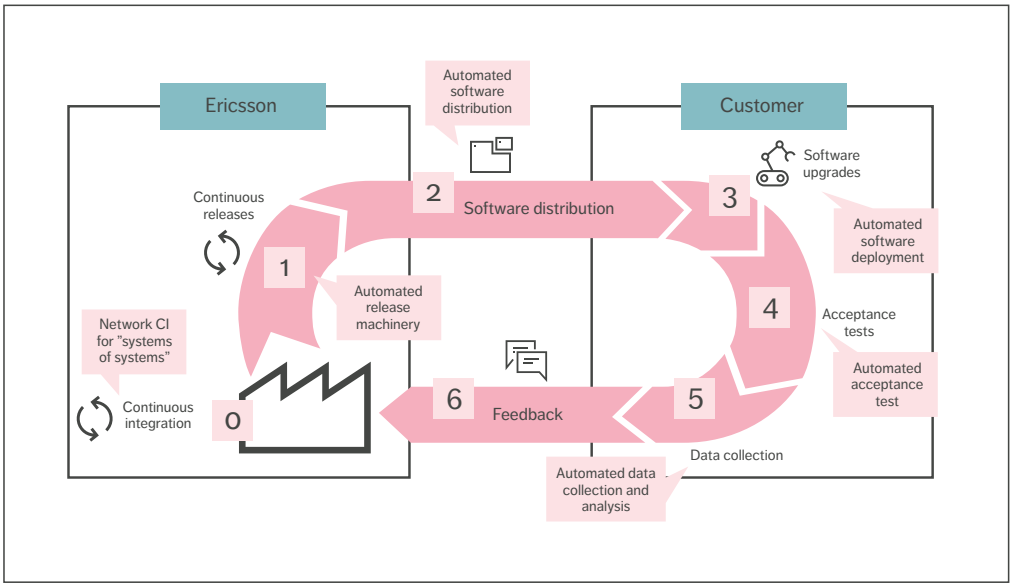


Figure 3 Fully automated CI/CD&D

store and visualize logs, metrics, traces and other data points, such as Prometheus, Fluentd, Elastic Stack, Jaeger and Grafana.

Security is a vital component of cloud-native development. On top of adhering to the best practices and guidelines provided by prominent organizations such as CIS (The Center for Internet Security) and NIST (the National Institute of Standards and Technology), open-source software projects such as Keycloak and HashiCorp Vault can help CNAs deal with storage and provisioning, as well as the handling of identities, certificates and keys.

To break down and implement business logic using stateless microservices, CNAs typically need to rely on stateful backing services to store their data. The type of stateful backing service that is required depends on various factors, such as the type and format of the data (such as structured or unstructured), the amount of data, the intensity of read and write operations, CAP and ACID properties, and so on. A multitude of open-source projects aims to address these needs, including database technologies such as PostgreSQL, MariaDB, Couchbase, Redis, MongoDB, Cassandra, MySQL and Hadoop.

The design philosophy behind Ericsson CNAs is to use polyglot persistence [4] while taking into account the total footprint and avoiding technology sprawl. Achieving the latter requires the identification of the most important properties that enable classification of database engine types into distinct groups and adopting a slightly opinionated approach in selecting one or a few choices in each group.

Continuous Integration, Continuous Delivery and Deployment

Our framework provides tools, interfaces and design rules that enable microservices to benefit from a fully automated Continuous Integration, Continuous Delivery and Deployment (CI/CD&D) pipeline, as illustrated in *Figure 3*. The pipeline is triggered from the moment code is committed and takes the new "candidate release" through the full cycle of build, verification, packaging and release. The deployment

phase will be somewhat different for aaS and aaP models. To get the full benefit from cloud native in an aaP approach, the end-to-end pipeline should be fully automated, connecting Ericsson with the customer to allow continuous feedback from live deployments to development teams.

> EXPOSURE IS LIMITED TO THE INTERFACE NEEDED BY THE USER OF THE SERVICE

Software reuse and open-source projects

While the goal of large-scale software reuse and utilization of open-source projects existed well before the emergence of the cloud-native paradigm and MSA, it is much more likely to be achieved now. This is because container technology isolates the different services from each other to a very high degree. Instead of being exposed to all the dependencies of each service, the exposure is limited to the interface needed by the user of the service.

Strict backward-compatibility of the exposed interfaces is required to achieve loose coupling and make it possible for each service to evolve independently. At the same time, semantic versioning on the interfaces creates a common way to communicate the evolution of the interfaces.

The CNCF provides a new starting point when looking for reuse opportunities from a cloud-native and MSA perspective. It adds value by creating a structure, choosing relevant open-source projects, and ensuring overall quality and community acceptance. It also provides a comprehensive map of potential realizations for different areas.

Key architectural aspects to consider

Making the right selection between the various CNCF and other open-source projects that are available requires a clear view of the context and the kind of use cases a selected service must support. Having a clearly defined architecture – with set goals, principles, design rules and guidelines – is more important than ever before in this situation because it helps to determine the different service and functional needs.

Following this approach, it is possible to disconnect the definition of which use cases are to be provided for within a particular area (and what additional principles would apply) from a particular realization. The benefit of this is that the architecture itself is not compromised or influenced by the need to support particular use cases (or not). For example, in the case of service mesh, this approach makes it possible to identify the core functional use cases that would add value to the architecture for CNAs, without being influenced by realizations like Istio and Linkerd.

Using identified use cases when considering different potential realizations both within and outside of CNCF enables a direct comparison when looking at the compliance to these use cases. This approach allows for reevaluation of previous selections for whatever reason. From an architecture evolution perspective, it is equally important to update identified use cases as soon as new needs arise, which may in turn lead to a new realization selection.

Separation of concerns

Separation of concerns is an important architectural principle that increases the possibility to reuse CNCF and other open-source projects even for very domain-specific use cases. In this context, separation of concerns means that the internal representation of the data should be separated from external representation. This approach makes it possible to apply cloud-native approaches in areas that were driven by purely proprietary implementations in the past.

Demonstrating the additional benefit and the potentially richer feature set offered by these projects provides an opportunity to influence and evolve expectations within the telecom domain. The ongoing evolution of ONAP (the Open Network Automation Platform) and support for high volume stream data collection is one example of this.

It is rare for out-of-the-box solutions from CNCF and other open-source projects to be sufficient to meet the needs of many telecom-specific use cases related to 3GPP, ETSI and other telecom-defined interfaces. There is, however, an obvious benefit to minimizing in-house development of the additions by building them as modular services on top of an available open-source project, when one exists in the relevant area. Realizations that are relatively generic and provide backward-compatible and version-controlled interfaces are preferred because they make it possible to build extensions in a future-proof way.

An example of this approach would be to choose Prometheus as the performance management infrastructure and use its interfaces to consume metrics and build additional support to expose 3GPP-compliant metric report files.

The question of maturity

Another important factor to consider when selecting an open-source project is maturity. This can be measured in terms of the size of the project's community and the number of releases it has that include backward-compatible changes. Selecting a mature project that is fully compliant with the use cases you have in mind would make it possible to take a passive role in the project, simply using the releases as they become available and focusing on building internal competence. Choosing a less mature project that is not fully compliant with your use cases would require you to adopt an active role in the community to influence both how backward-compatibility is managed and the direction of feature evolution in the future.

In either case, it should be noted that using open-source software is not free – it is important to build up internal knowledge related to the software, especially around the needed use cases. It is not an option to be dependent on the community for all support-related questions, particularly when taking on an active role.

Be aware that, during the project selection process, it will typically only be possible to get a snapshot view of how the project handles the critical issues of backward-compatibility on exposed interfaces and semantic versioning. While a project with a longer history should be able to provide a better picture, it still might not be able to offer a complete one. Kafka is an example of a relatively mature project in which changes that broke backward-compatibility (according to our definition) were announced as a minor update in the release, rather than a major release.

> IT IS IMPORTANT TO BUILD UP INTERNAL KNOWLEDGE RELATED TO THE SOFTWARE, ESPECIALLY AROUND THE NEEDED USE CASES

Weighing integration potential against best-of-breed

In several cases, open-source projects have been created with a very different context in mind than the one in which they are later used within a defined architecture – particularly in terms of deployment footprint and characteristics aspects. For example, a project may decide to use one or two specific data stores for persistent data, which, given the bigger picture of the CNA architecture, may not be natural fits.

Sometimes a choice like this is made with an aaS context in mind, where a service is deployed once and reused by everyone, as opposed to an aaP context, where a CNA typically needs to be more self-contained and provide its own instances of all services. One example of this is Harbor, a project that has an opinionated selection of both data store and ingress. In many cases, these sorts of issues can be addressed by configuration or influencing the project, but sometimes they lead to a different selection of realization.

As each open-source project is typically independent, functionality overlap is common,

and there can sometimes be contradicting views about how particular problems should be solved. While each project can typically provide value within the scope it was designed for, the broader architectural perspective requires the provision of end-to-end value without compromising defined goals and principles. A key aspect of this is figuring out how different projects can be integrated and used in combination to provide greater value.

In light of this, it is important to keep in mind that even though a specific open-source project may be considered best-of-breed, it might not fit well into the full end-to-end value that the larger architecture intends to provide. In this case, it might make more sense to select a less capable project that is a better fit with the overall architectural goals.

For example, existing best-of-breed projects would not be the right choice to build a cohesive, fully integrated visualization solution to provide a high-level view of microservices and their health status, metrics, logs, distributed traces and other artifacts needed for monitoring in a DevOps team. Bundling best-of-breed standalone user interfaces (UIs) such as Grafana for metrics, Kibana for logs, a Jaeger UI for traces and Kubernetes UI for workload monitoring would result in very poor usability and a sub-optimal experience for Ops teams.

> DIFFERENT PROJECTS CAN BE INTEGRATED AND USED IN COMBINATION TO PROVIDE GREATER VALUE

Shared responsibility for security

Certain aspects of security are expected to be covered by open-source projects, while others remain the full responsibility of the development organization. For instance, when it comes to securing communication, there is typically an aligned approach within both the enterprise and telecom domain. This is centered around TLS and OAuth2, especially looking at the evolution of 3GPP, which is moving to web-scale technology protocols like HTTP/2. When this level of security is not provided by the open-source project, it is typically seen as a valid evolution of the project. In some scenarios, though, due to timing or conflict between free and commercial versions of the project, the mitigation is to deploy a proxy in front of the service to address security aspects – albeit at the cost of introducing additional latency.

The model for using open-source software is to bring in the source code – even if pre-baked container images are provided by some projects – and build container images, including the selection of the base operating system image. Because of this, security hardening must be fully controlled by the team. There are standard testing tools such as CIS-CAT (the CIS configuration assessment tool) that help teams evaluate the quality of the hardening they have performed. Such reports can also be used to assure customers of the overall security compliance.

Key organizational and cultural aspects to consider

Speed is the driving force in the cloud-native paradigm, which means that the goal is to streamline the work process as much as possible. Every task that cannot be automated or cut out inhibits the organization's ability to benefit from the cloud-native approach. Moving to the cloud-native paradigm and making use of MSA is therefore much more than just a technological change in how software is built. A number of important organizational and cultural changes must take place to achieve the benefits.

A cloud-native development organization is often structured around architecture and processes. When the need to create a service has been identified, a small team forms to take full responsibility and accountability for that service across all stages of the software life cycle, according to DevOps principles. This structure stands in stark contrast to the traditional one, in which bigger teams typically work on larger software projects alongside teams with dedicated responsibility for horizontal tasks such as release and compliance handling.

Moving from a more traditional software release cycle of every three to six months to much more frequent releases requires both a higher level of automation of the process and a reduction in the number of activities/tasks needed as part of a release. In short, anything that can be automated must be automated, and a few tasks, such as trade compliance, that cannot be automated must be simplified.

> TO FULLY BENEFIT FROM THE CLOUD-NATIVE PARADIGM, SPEED CANNOT BE COMPROMISED

Building trust and acceptance

Developers do not have full control when they work with open-source code. This can lead to trust issues, particularly in organizations that have a history of working with proprietary implementations. Even when there is a buy-in from developers on the decision to use open-source code, it is often necessary to build trust and acceptance within the organization for the fact that open-source code can do as good a job of fulfilling requirements as code that is developed in-house.

The best way to back up the decision to use open-source code is to demonstrate that there is a solid selection process in place along with clear mapping of architecture use cases. From a business point of view, it is important to emphasize that the cost of developing commoditized software is simply not justifiable: selecting open-source software that meets the criteria to a sufficient degree is preferable to creating similar software from scratch.

Regardless of whether a service is based on open-source code or a proprietary implementation, the team structure and the responsibility must be the same. Constant validation of the open-source project is crucial for early detection of any potential issues relating to backward-compatibility.

In-house talent is crucial to the successful use of open-source code. Internal competence building is as important for open-source based work as it is for proprietary implementations, both for troubleshooting purposes and to understand evolution needs.

Conclusion

Cloud-native application design will soon become the telecom domain's new standard for the development of virtual network functions. Ericsson is well prepared for this transition, thanks to our application development framework, which uses cloud-native principles, microservice architecture and several key enabling technologies such as containers and Kubernetes.

To fully benefit from the cloud-native paradigm, speed cannot be compromised. Technological and architectural changes are not enough – transitioning toward the cloud-native paradigm requires major organizational and cultural changes as well.

Achieving the necessary speed requires smaller team setups with full DevOps responsibility and (full) automation of any step that is required as part of the software CI/CD&D and release process. One of the main organizational challenges is to get the developers to let go of full control and trust the use of open source in areas where in-house development has been used in the past.

THE AUTHORS

Henrik Saavedra Persson

◆ is an expert at Business Area Digital Services, where he drives the common cloud-native architecture for the business area's virtual network functions and applications in his role as chief architect of the application development framework. He joined Ericsson in 2004 and has worked in R&D with architecture for both applications and platform products. Saavedra Persson holds an M.Sc. in software engineering from Blekinge Institute of Technology in Sweden.

Hossein Kassaei

◆ is a technology specialist in cloud computing and MSA within Business Area Digital Services, where he has worked on common platforms and application architectures for the past five years. He joined Ericsson in 2010 and has worked in various roles within R&D from developer, to software designer, to architect and DevOps lead. He holds an M.Sc. in computer science from Concordia University in Montreal, Canada.

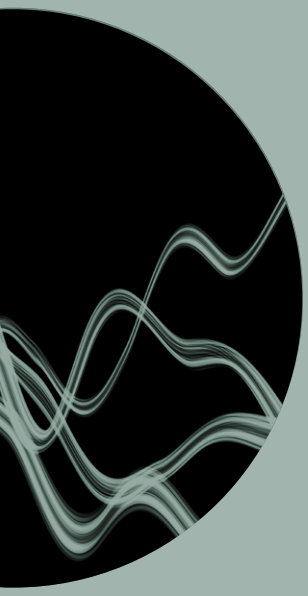
The authors would like to thank Mario Angelic for his contribution to this article.

References

1. **Ericsson, Cloud-native applications, available at:** *https://www.ericsson.com/en/digital-services/trending/cloud-native*
2. **CNCF, Certified Kubernetes, available at:** *https://www.cncf.io/certification/software-conformance/*
3. **GitHub, CNCF landscape, available at:** *https://github.com/cncf/landscape*
4. **Martinfowler.com, PolyglotPersistence, November 16, 2011, available at:** *https://martinfowler.com/bliki/PolyglotPersistence.html*

Further reading

» **CNCF, Sustaining and integrating open source technologies, available at:** *https://www.cncf.io/*



ISSN 0014-0171
284 23-3329 | Uen

Ericsson
SE-164 83 Stockholm, Sweden
Phone: +46 10 719 0000