ScaleIT Platform Documentation Documentation

Release 1.0.0

Andrei Miclaus

31.12.2018

Contents:

Bemerkung:

Warum ScaleIT? *Schauen Sie sich unsere Value Proposition an!*

Was ist ScaleIT? *Lernen Sie was ScaleIT ist*

Bemerkung: Click here for the technical getting started (GitHub)

Click here for the architecture documentation and white paper

KAPITEL 1

What is ScaleIT?

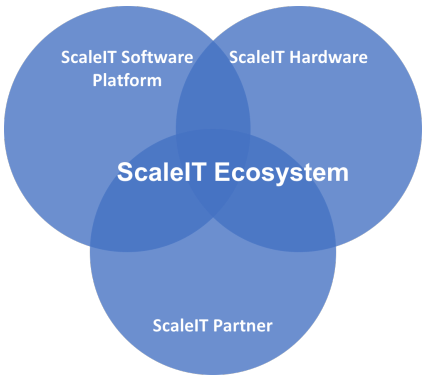
ScaleIT is a solution to bring advanced IT technologies to manufacturing companies. We bring the necessary **software and hardware infrastructure** to the manufacturing shop floor of small and medium companies and help people work better by introducing **modern Industrial Grade Apps**.

It is a work in progress created within the ScaleIT Project funded by the BMBF (Fkz.: 02P14B180ff).

Bemerkung:

ScaleIT Value Proposition *Ich möchte jetzt erfahren wie mir ScaleIT auf dem Hallenboden hilft!*

There are three main components to the ScaleIT solution the ScaleIT Software Platform, ScaleIT Hardware and the ScaleIT Community, as seen in the figure below.



ScaleIT Software Platform In IT, a platform is any hardware or software used to host an application or service. An application platform, for example, consists of hardware, an operating system and coordinating programs that use the instruction set for a particular processor or microprocessor[1].

Have a deeper look at the software architecture *here*.

[1] https://searchservervirtualization.techtarget.com/definition/platform

ScaleIT Hardware WIP, coming soon

ScaleIT Partner and the ScaleIT Community Meet the people of ScaleIT

Abb. 1: Das ScaleIT Konsortium

ScaleIT is Open Source When a project is open source, it means anybody can view, use, modify, and distribute your project for any purpose. These permissions are enforced through an open source license.

Learn why this is important OSS for German Manufacturing

ScaleIT is being published with an MIT license and are provided as is.

If you need professional solutions, our partners are building industry grade ScaleIT Systems and are happy to help. Mail us at info@scale-it.org or e-mail a partner directly.

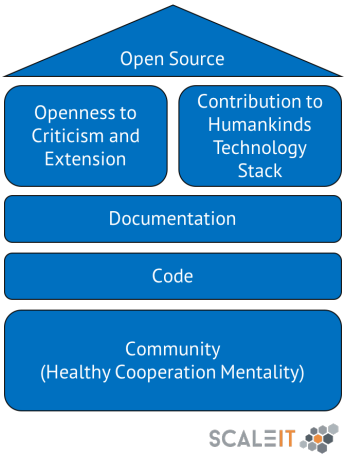


Abb. 2: Pillars of open source projects.

Zu tun: Better image with ScaleIT Eco System, ScaleIT Software Platform, ScaleIT Hardware

KAPITEL 2

Get Started

2.1 Technical Getting Started

 Click here for the technical getting started (GitHub)

 *Click here for the architecture documentation and white paper*

2.2 Understanding the Shop Floor

IoT Aspects of the Shop Floor

Wo befinden wir uns wenn wir vom betrieblichen Hallenboden reden?

Production ready IoT systems require a sophisticated and robust infrastructure. There are a multitude of services and open source technologies that allow the realisation of such systems with minimal effort. In practice however, and especially in the manufacturing domain, it proves difficult to bring these technologies into production and as a result the shop floor cannot benefit from the advantages of the IoT. Working in close cooperation with small and medium sized manufacturers, we have identified several key issues they face when dropping the monolithic approach and adopting IoT technologies and an IoT mindset:

- The shop floor systems are isolated from the Internet.
- System landscapes are dominated by unconnected silos or monoliths (Enterprise Service Bus, ERP Systems, large central data warehouses). These are difficult to evolve and do not fit with the distributed nature of the shop floor.
- Technology choices are limited to only a handful of in-house-approved stacks due to restricting corporate IT policies (e.g. Windows only world).
- It is hard to iterate through software versions due to difficult development and deployment and divergent infrastructure.

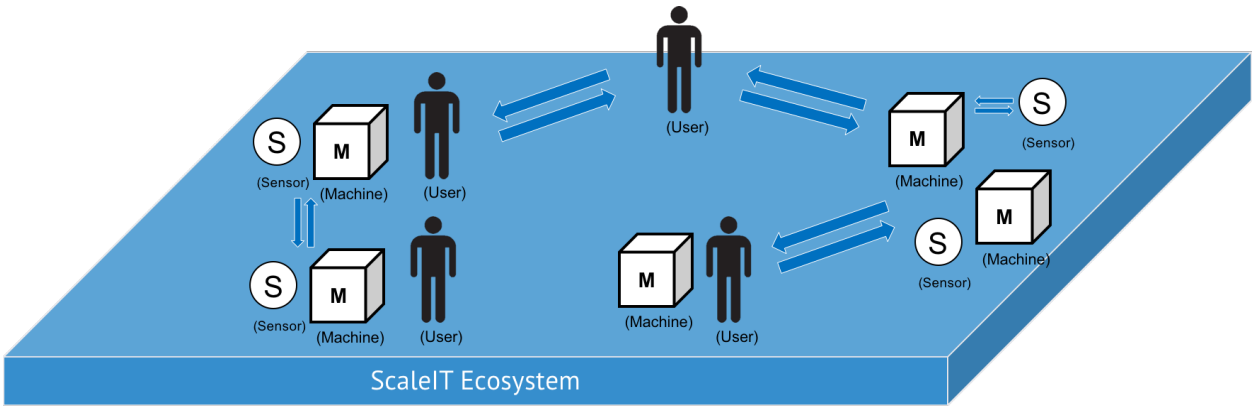


Abb. 1: The manufacturing shop floor where people, machine and sensors interact not just physically but also virtually through software.

- Slow, low risk approach to potential game changer technologies due to a (justified) high risk aversion regarding the control software for the production lines.
- Difficult testing and simulation using realistic or real data as it is either non existent or not easily reusable.
- It is generally hard to create a distributed interconnected environment having only a limited workforce consisting of small IT teams and no dedicated ecosystem software engineers.

By incorporating key software engineering concepts (versioning, containerisation, continuous integration, continuous deployment, App runtime environment) the proposed architecture aims at giving manufacturers a ready to use infrastructure with the ingredients for a healthy and flexible software eco-system on the shop floor. We believe that the design principles listed below are central to a shop floor architecture that supports IIoT applications:

- Support silo centerd architecture
- Resilience first design
- Rapid deployment

2.3 ScaleIT System Landscape

Using ScaleIT, you imbue the IT landscape on your manufacturing shop floor with the best practices from modern software development and can harness benefits such as faster innovation cycles, lower risk and increased productivity by using better tools.

References

Some icons provided by oNline Web Fonts

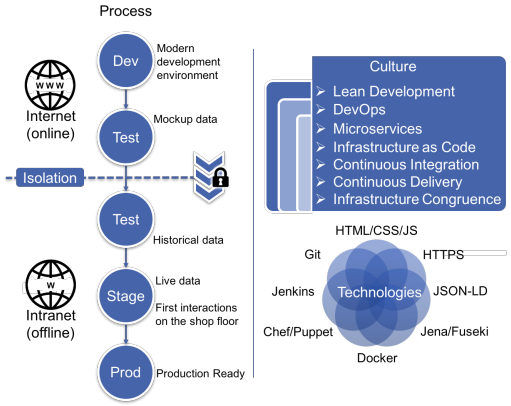


Abb. 2: The Web of manufacturing overview

KAPITEL 3

Value Proposition

- Schnell installierbar
- Integration in Firmen-IT
- Semantische Daten
- Hohe Integrierbarkeit durch Web-Technologien
- Kein Internet notwendig

3.1 Digitalisierung durch Apps

Bemerkung:

ScaleIT success stories hier! Sehen sie wie ScaleIT unseren Partneunternehmen geholfen hat.

3.1.1 ScaleIT Digitalisierungsstrategie

3.2 Anforderungsanalyse

If you wish to analyse your requirements regarding Apps, visit our ScaleIT App Workshop Kit: https://github.com/ScaleIT-Org/workshop-app-prototyping

3.3 Prozesse

Die Verbesserung von Prozessen auf dem betrieblichen Hallenboden ist eines der Hauptziele von ScaleIT. Unser Ansatz heißt *Die richtige Software für den richtigen Job einsetzen*.

Abb. 1: Apps bieten Nutzern einen hohen Wert durch die Einfachheit der Installation und Bedienung, sowie der optimierten nutzefreundlichkeit (UX).

Abb. 2: Zusätzlich zu der Bedienungsfreundlichkeit von Smartphone-Apps, bieten die Industriellen Apps von ScaleIT weitere Vorteile und können im betrieblichen Kontext eingesetzt werden.

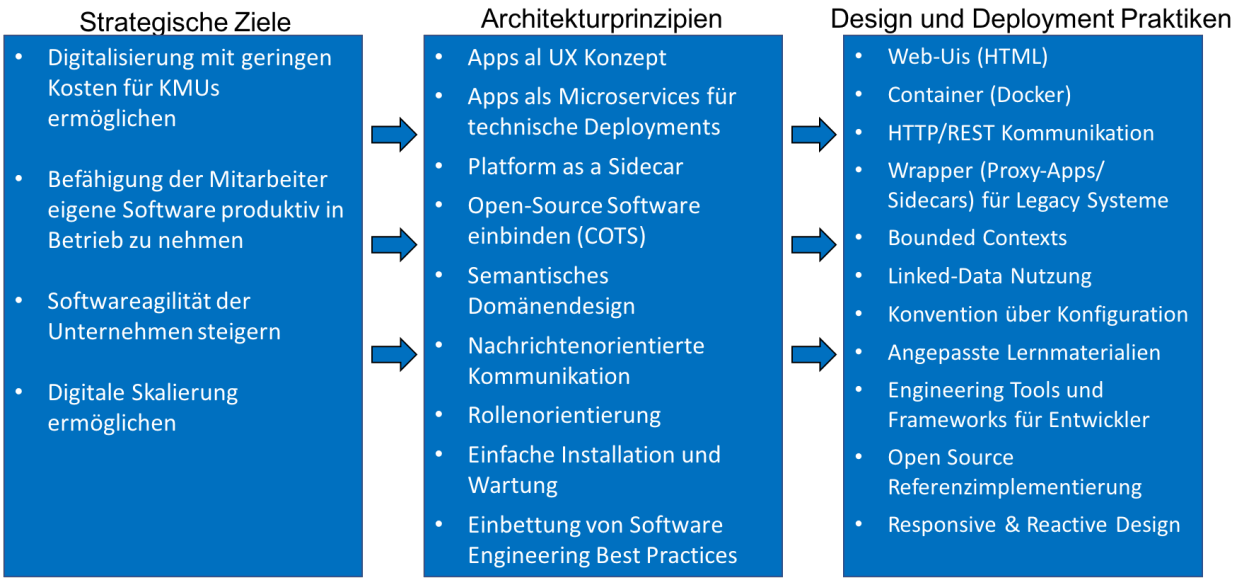


Abb. 3: Digitalisierungschritte durch ScaleIT: von der Strategie hin zu den konkreten Maßnahmen.

3.3.1 Software auf dem Hallenboden risikolos ausprobieren

Durch das Vereinen von Development und Operations (Entwicklung und Inbetriebnahme) wird es Experten ermöglicht Software leicht auf den Hallenboden einzusetzen oder zurückzurollen. Da Apps Seiten-Effekt frei sind, können nach dem Baukastenprinzip Apps entfernt werden.

3.3.2 A/B Testing

Durch die Vereinfachung der Installation von Software durch ScaleIT, wird es mittelständischen Betrieben ermöglicht unterschiedliche Apps auszuprobieren und dann die geeignetste App zu behalten.

Sucht ein Unternehmen beispielhaft ein Projektmanagementwerkzeug, dann kann es die Redmine-App, die Gitlab-App oder eine andere App aus dem Katalog installieren und diese parallele betreiben und ausprobieren.

Die Entscheidung wird so durch reelen Einsatz der Software im betrieblichen Alltag unterstützt.

[1] https://commons.wikimedia.org/wiki/File:A-B_testing_example.png
[2] https://commons.wikimedia.org/wiki/File:Ab-test.jpg

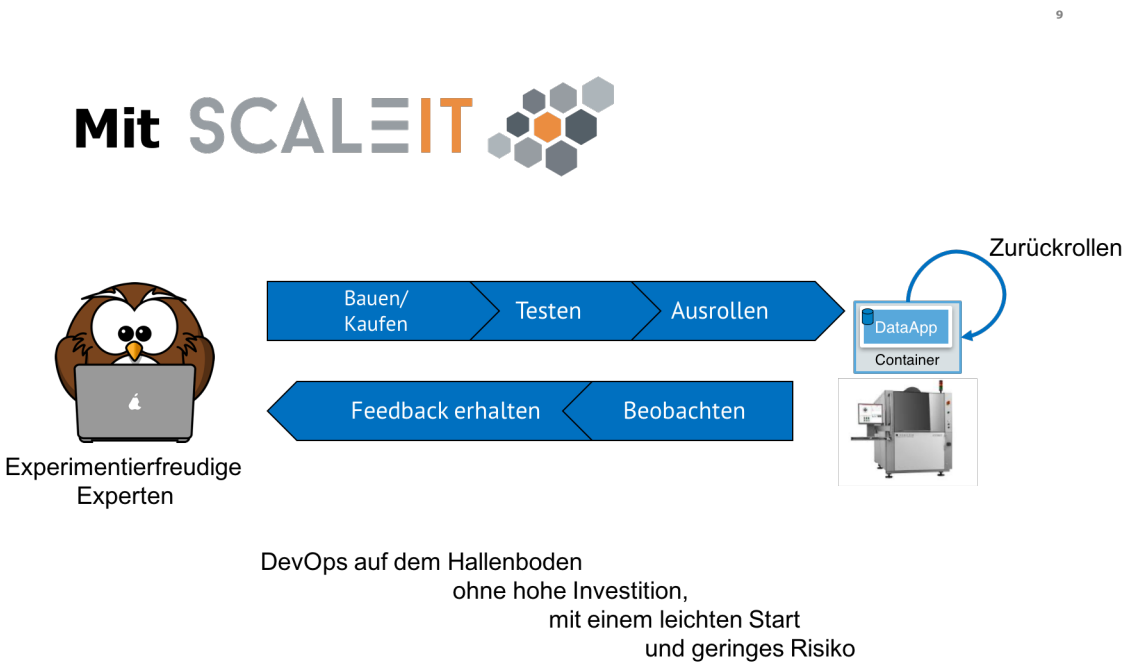


Abb. 4: In ScaleIT ist der Lebenszyklus eines Stückes Software (der App) vereinfacht.

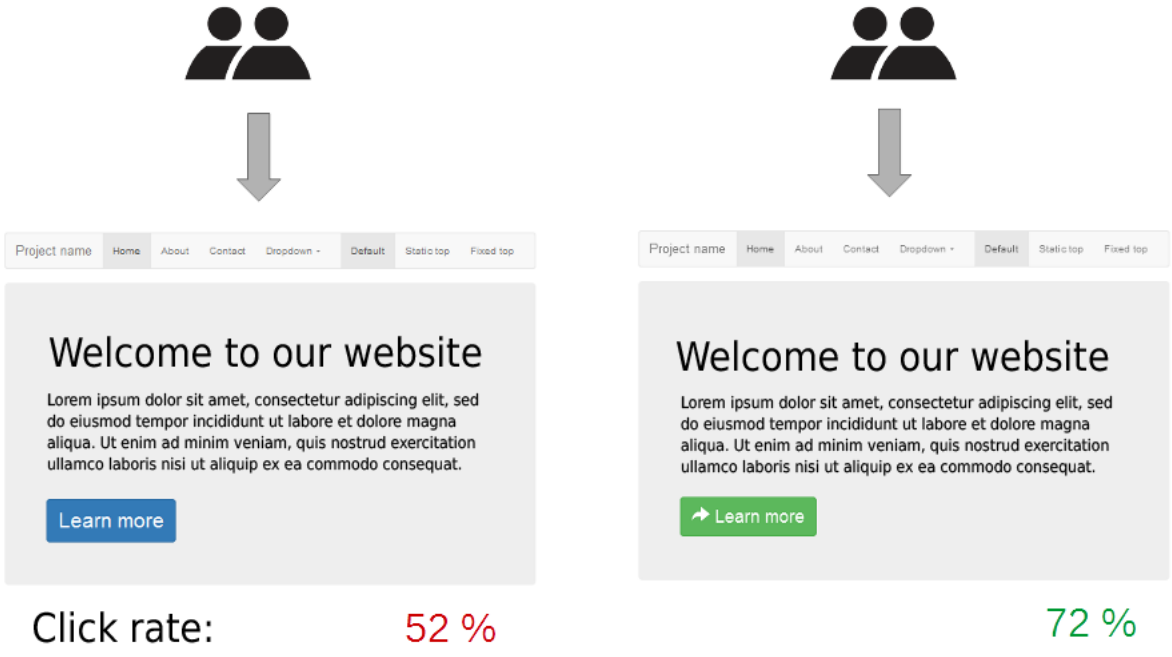


Abb. 5: A-B Test Beispiel[1]

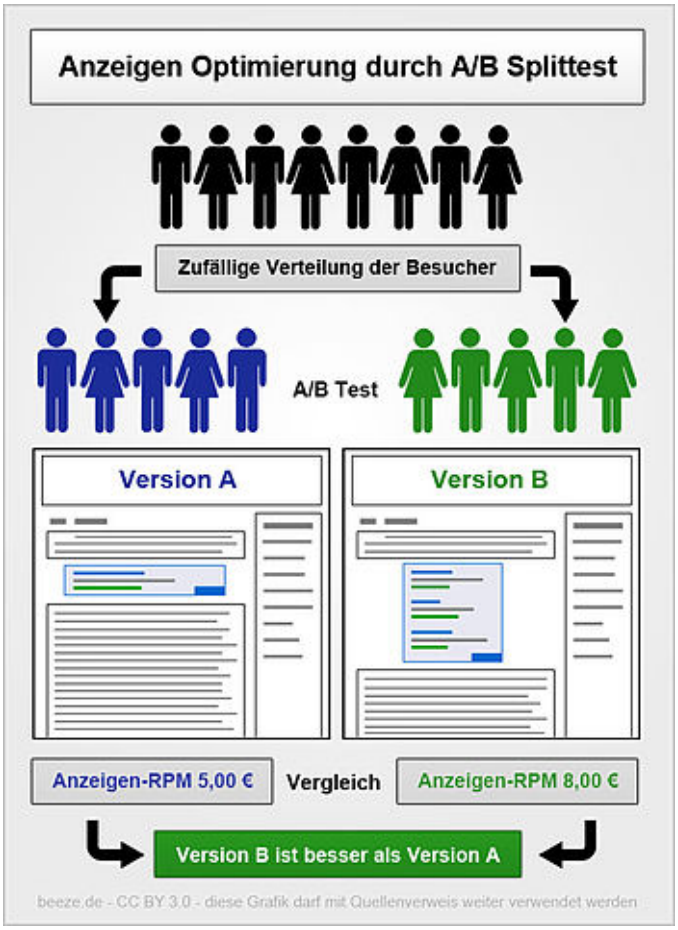


Abb. 6: A-B Test Beispiel[2]

KAPITEL 4

Skalierung innerhalb ScaleIT

Liste der Skalierungs-KPIs für die ScaleIT Plattform und das ScaleIT-Ökosystem.

4.1 Skalierung auf der Firmen-Ebene

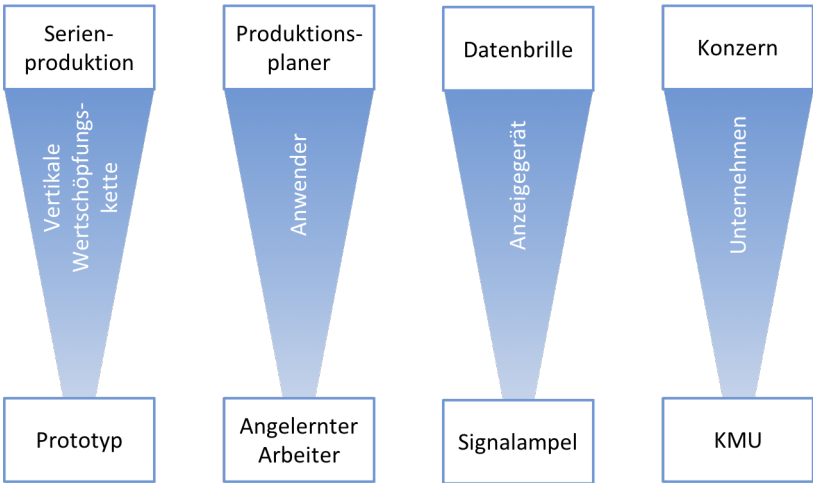


Abb. 1: Die Hauptaspekte der skalierbaren IKT zur Nutzung auf dem Hallenboden.

4.2 Plattform Skalierung

Die hier vorgestellten theoretischen Skalierungsebenen der Plattform dienen als Grundlage für die organisation und das Ausrollen der ScaleIT-Plattform-Lösung bei Firmen von unterschiedlichen Größen und IT-Kompetenzen.

Die Skalierungsdimensionen sind (a) das Level an Plattformfunktionalität (Iaas-SaaS), (b) die verantwortlichen Entitäten für das Vorantreiben der Entwicklung (Forschung, Entwicklung, Systemhäuser, Kunden), (c) die Komplexität der

Lösung mit Hinblick auf die vollständige Integration in die Unternehmenslandschaft (Libraries, Engineering Tools, einheitliche Sicht).

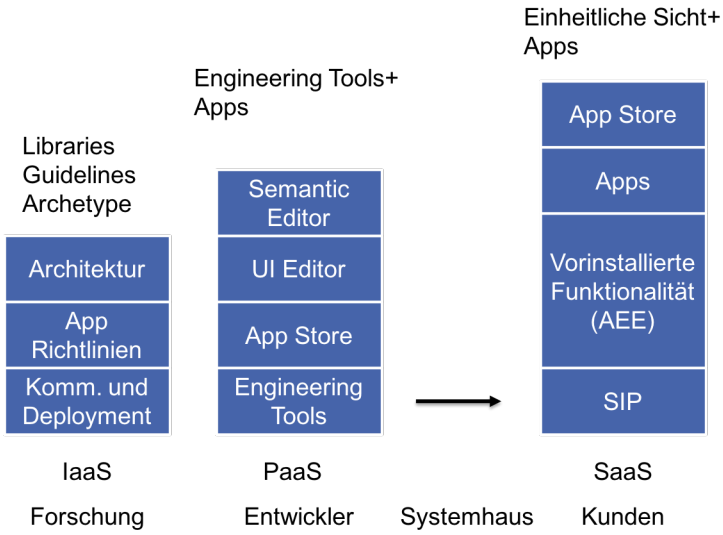


Abb. 2: Die Skalierung der Plattform bezüglich auf die Plattformfunktionalität, der Weiterentwicklung und der Integration in die Unternehmenslandschaft.

4.3 Architekturell-technische Skalierung

Die technische Skalierung weist eine hohe intrinsische Komplexität auf. Das Ziel der ScaleIT-Architektur ist es die extrinsischen Faktoren zu minimieren. Der Mensch muss weiterhin als kreatives Element einbezogen werden, um logisch sinvolle Anwendungsnetzwerke und vernetzte Dienste zu ermöglichen. Die von ScaleIT bereitgestellten Technologien, Konzepte und Paradigmen bieten hohe gestalterische Möglichkeiten ohne Nutzer und Entwickler unnötig zu belasten.

Software-Technisch/Architekturell

Microservices Die Software-Organisation von ScaleIT beruht auf den Prinzipien der Microservice Architektur (die moderne Reinkarnation der SOA - Service Oriented Architecture). Dadurch bietet sich die flexible Organisation und Reorganisation der Software auf den betrieblichen Hallenboden.

Lambda Innerhalb der ScaleIT-Architektur kann eine Lambda-Architektur abgeblidet werden. Die Apps müssen ledliglich in die jeweiligen Batch und Speed Layer eingeordnet werden. Hier bietet sich die Option an, die Layer der Lambda-Architektur auf unterschiedliche Hosts aufzuspalten. Alternativ kann ein System wie z.B. Spark direkt als App laufen (dabei ist jedoch zu beachten, dass solch komplexe Apps am besten auf einem separaten Host mit ausreichend Ressourcen und Monitoring ausgefürt werden).

Kappa Die Kappa-Architektur ist die von ScaleIT empfohlene Software-Organisation für Szenarien die auf Datenerhebung und -analyse zielen. Besonders bei Implementierungen mit digitallen Zwillingen kann durch die lokale Speicherung der Daten von der jeweiligen Maschinen-App eine einfache und robuste Implementierung realisiert werden.

FaaS - Function as a Service Wenn es um das Ausrollen von kleinen in sich geschlossenen Funktionalitäten geht, so bietet sich innerhalb der ScaleIT-Architektur die Option eine FaaS-App einzuseten. Dies ist besonders sinnvol, um eine automatische Skalierung bei wiederholten Aufrufen einer einfachen ausführbaren Logik (wie z.B. das Auswerten eines ML-Modells auf Anfrage über eine HTTP/REST Schnittstelle). Das

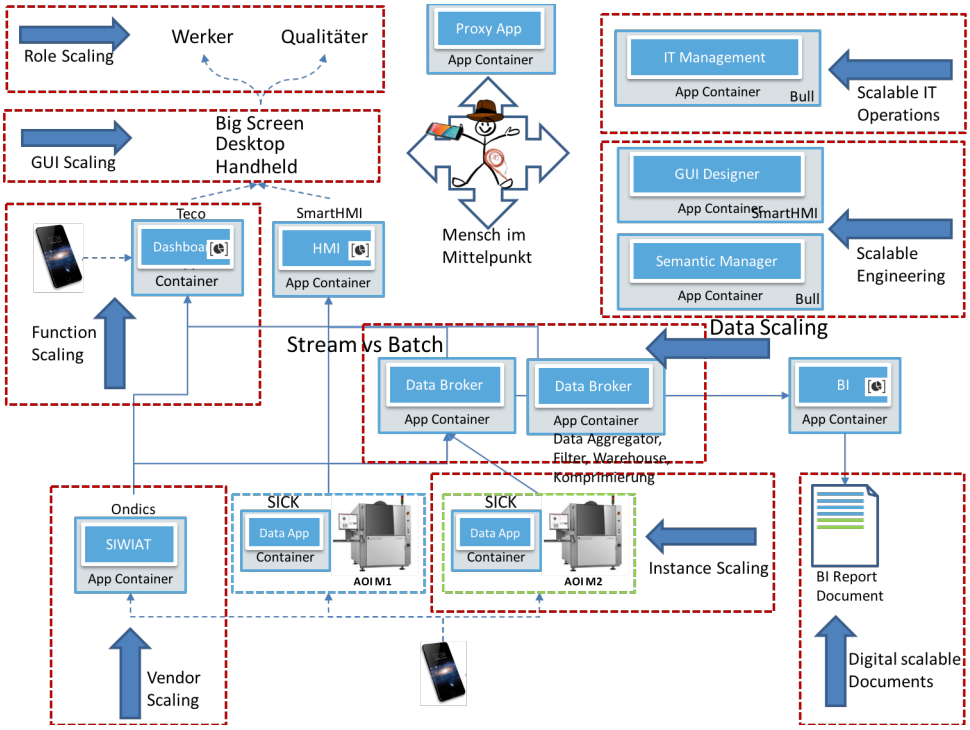


Abb. 3: Überblick der technischen Kernskalierung dank der ScaleIT Architektur

Erstellen multipler Apps für solch kleine Funktionen ist nicht zielführend und wirkt sich negativ auf die Wartung der Instanz.

Programmiersprachen Polyglote Stacks sind in ScaleIT möglich und dienen auch der Funktionsskalierung und der horizontalen sowie auch der vertikalen Vernetzung. Ein zu breites Spektrum an Technologie-Stacks sollte aber vermieden werden, um den Wartungsaufwand klein zu halten (siehe auch Microservice Best Practices).

Kommunikation

Datentransfer über Web-Technologien Kommunikationsmechanismen die in ScaleIT App eingesetzt werden können sind folgende:

REST/HTTP (Empfohlen)

- Request Reply und Polling über HTTP Methods (GET, POST, PUT, DELETE)
- Streams über Server Sent Events (SSE)

GraphQL (Experimentell)

- Graphbasierte abfragen über das Web mit Pagination Support

MQTT

- QoS gesicherte Streams

WebSockets

- Payload-agnostisches Protokoll ohne Transfersemantik

RTC

- Real Time Web Kommunikation (z.B. Kamera-Streams)

gRPC/Protocol Buffers

- Hocheffiziente Remote Procedure Calls über das Web (z.B. auch über Captain Proto)

Stream vs Poll vs Batch Durch die unterschiedlich unterstützten Kommunikationsmechanismen ist es möglich eine für den Anwendungsfall optimale Lösung zu finden. Für Datengetriebene Anwendungen werden bei ScaleIT Streams über SSE und für Polling(Request/Reply) HTTP empfohlen. Zu beachten ist, dass HTTP schon eine Semantik mitbringt, MQTT und WebSockets nicht. MQTT eignet sich aber sehr gut, um den Nachrichtenaustausch bezüglich der Zustellung zu sichern und bietet dazu 3 Level an Quality of Service (QoS).

Payload Skalierung In ScaleIT ist das empfohlene Datenformat der Payload eine semantische Darstellung in JSON-LD. Es können jedoch auch andere Formate genutzt werden (wie z.B. JSON, Protocol Buffers, proprietäre Formate etc.). Dies ermöglicht eine Skalierung bezüglich der datengetriebenen Interaktion zwischen Apps, vorausgesetzt die Apps unterstützen diese Datenformate. Nutzt man eine semantische Darstellung, dann wird. Durch JSON-LD(und auch JSON) können auch inkomplette Nachrichten verschickt werden, ohne das Anwendungen abstürzen und diese auch nur Teilverarbeitung ermöglichen.

Migration auf andere technische Plattformen Durch die Containerisierung der ScaleIT Apps und des dazügehörigen Infrastructure as Code (IaC) Paradigmas ist die Migration von ScaleIT Software auf andere Plattformen sehr effizient. Der Kern der App Architektur bildet das Prinzip des Platform as a Sidecar, so dass Apps mit einer minimalen Konfigurationsänderung auf einer anderen Instanz laufen können oder ein Wechsel weg von ScaleIT (z.B. auf öffentliche oder private Clouds) möglich ist. Durch die Lokalität der Daten innerhalb der Apps, sind bei korrekten Implementierungen der Plattform keine Datenmigrationen notwendig. Auch aus diesem Grund ist keine zentrale Datenhaltung empfohlen (außer Backup und Archivierung).

Funktionalität von Dritten Da KMUs meistens keine eigene Software-Entwicklung betreiben erlaubt es die modulare ScaleIT-Architektur Software (Apps) von Drittanbietern und Dienstleistern einzubinden. Durch das technologieagnostische Design können auch unterschiedliche Dienstleister Software für dieselbe Firma aktiv werden.

Dokumentation

Daten

Funktionsskalierung Die in der ScaleIT-Architektur unterstützen Paradigmen Container, Microservices und FaaS erlauben ein Reibungsloses Klonen von schon vorhandenen oder die Integration von neuen Funktionalitäten.

Egineering(Tools)

Management Ein zentralisiertes Management ist möglich in der ScaleIT-Architektur und in größeren Instanzen empfohlen. Durch das installieren der passenden Apps (Kubernetes) kann z.B. das Container-Managment automatisiert werden.

Monitoring & Logging Ein zentralisiertes Management ist möglich in der ScaleIT-Architektur und in größeren Instanzen empfohlen. Durch das installieren der passenden Apps (ELK Stack) kann z.B. das Logging zentralisiert und durch Dashboards auch für den Nutzer zusammengeführt werden.

Nutzeröberflächen Die in ScaleIT genutzten Web-Technologien erlauben Responsive Design und Adaptive Design anzuwenden, um die Oberflächen der Apps auf unterschiedliche Geräte zu skalieren. Z.B. kommt das in der Open Source Referenzimplementierung empfohlene Ionic-Framework mit Responsive Design out-of-the-box.

Rollen Die Mitarbeiterrollen in einem Unternehmen sind ein wichtiger Aspekt der organisationellen Kultur. Die ScaleIT-Architektur erlaubt es Apps (u.a. auch Open Source Apps) zu nutzen, um unterschiedlichen Unternehmensgrößen und Bedarfe zu decken. Gekoppelt mit dem Singel-Sign-On Mechanismus (SSO Manager + SSO Sidecars) werden die Rolleninformationen allen Apps die damit umgehen können bereitgestellt. Auch kann dieser Mechanismus nachgerüstet werden und muss nicht in der Intialinstanz vorhanden sein.

Development & Operations & Agilität Die ScaleIT-Architektur ist für alle modernen Arbeits- und Entwicklungsprozesse ausgelegt. Durch das installieren der passenden Apps können z.B. auch DevOps oder DevSecOps Anwendung auf den betrieblichen Hallenboden finden.

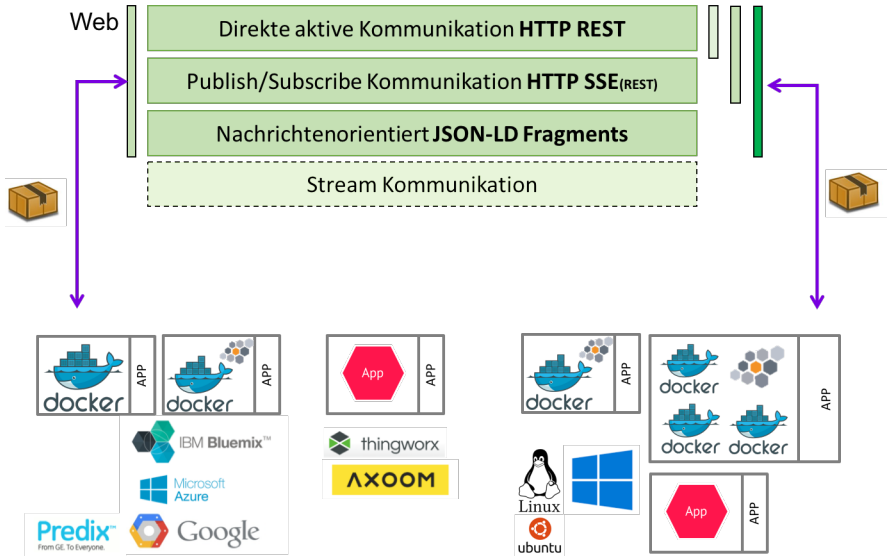


Abb. 4: Die Skalierung der Kommunikation bezüglich auf die Art des Transfers.

KAPITEL 5

ScaleIT Success Stories

Zu tun: Publish success stories.

5.1 Shopfloor Monitoring bei der SICK AG

Zu tun: Upload KIT + SICK poster

Involved partners: Ondics GmbH, Worldline, SICK AG, KIT

Results: Better understanding of live data and optimization potential

Electronic manufacturing is a sensitive process in which humidity and temperature, for example, play an important role. To ensure quality, optical inspection steps are performed on every printed circuit board (PCB).

However, under certain conditions, issues with the PCBs may still arise. Therefore, to improve the quality and throughput, the shop floor has to be continually upgraded with new capabilities that allow employees to promptly react and reduce errors.

First, employees need to visualize the quality inspection data, both on large screens and their smartphone. This in turn allows them make timely and more informed decisions. Second, temperature and humidity sensors need to be installed in order to monitor the production environment. In case some sensors use different units (C vs. F), they still need to present meaningful data to workers or other machines. Third, inspection, temperature and humidity data need to be aggregated for both workers and data analysts. The latter need to be able to use the data in order to train machine learning models for predictive maintenance.

The fourth and final step, is the deployment of the machine learning algorithm into production and associating the predictions with machine location and PCB meta-data. This scenario requires physical devices and software from different vendors that need to interoperate to achieve the desired outcome. While the predictive analytics step is still in development, the mashup is explained in greater detail in the following section.

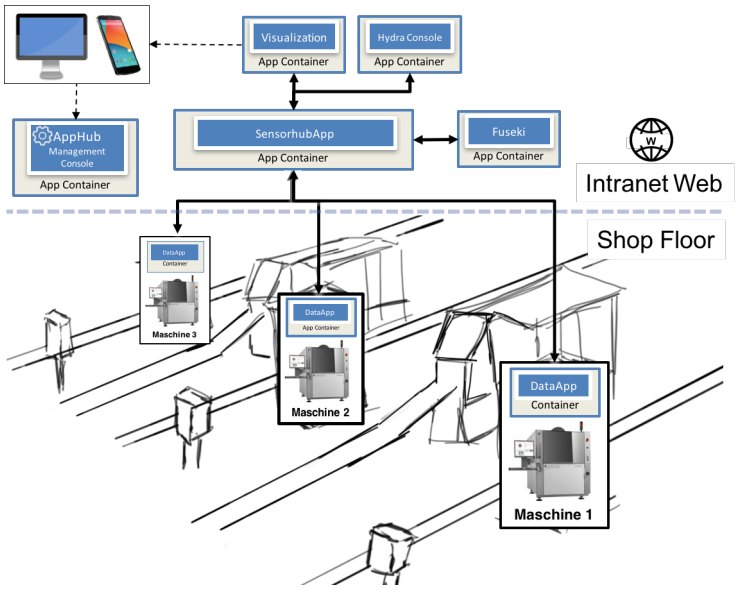


Abb. 1: Anwendungsszenario der Apps bei der SICK AG

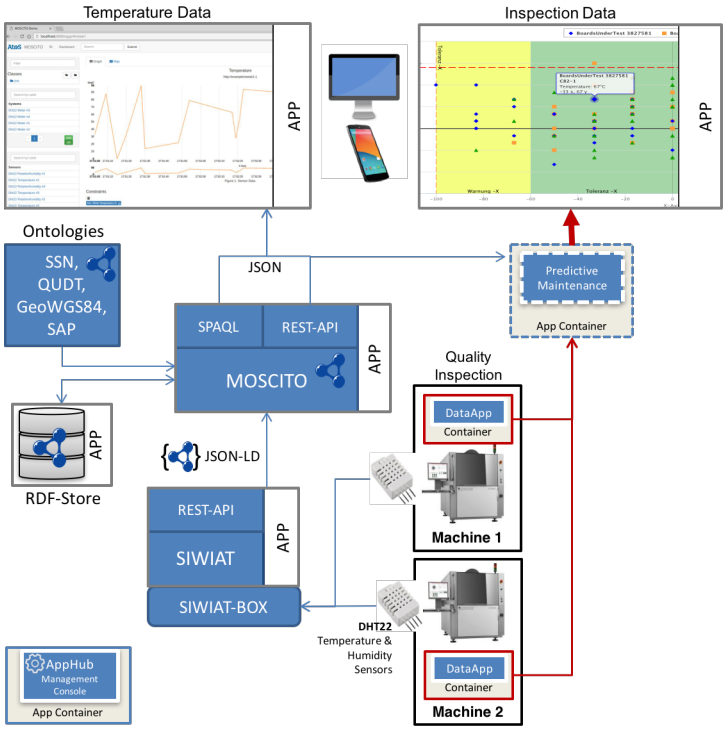


Abb. 2: Gesamtüberblick des implementierten Szenarios zu Sensordatenfusion

The proof of concept mashup shown in the figure above, consists of apps that are implemented as stand-alone Docker containers with functionality restricted to a small, business value generating subset. Every app is uniquely identifiable and addressable within the web by its URL and RESTful interactions are possible.

The apps have been developed independently and we have relied on common semantics to allow their interoperation. The App Hub (lower left) is a simple app store. It manages each app's life cycle by generating Docker-Compose files according to which apps are part of requested mashups. SIWIAT3 is an embedded gateway-as-a-box with internal apps developed by the authors and targeted at the industrial shop floor. Its role is to wrap legacy or non-web-capable sensors, annotate the measured values and make them available via a REST interface.

MOSCITO4 is a semantic middleware developed by Worldline based on OSGI, Apache Jena, and the RDF4J framework. It provides a set of functionality for the management of semantically linked data accessible through RESTful Web Services, such as ontology and rule management, and, moreover, SPARQLEndpoints as well as triple store connections and data integration from heterogeneous data sources. MOSCITO acts as a shop floor semantics engine and a visualizing tool that collects and connects related data form heterogeneous data sources but also manages ontologies and rules.

Machine 1 and 2 are automatic optical inspection (AOI) machines. Inspection data is available via publish/subscribe through the data app (using server-sent events). Adding another machine to the mashup requires just the installation of the data app for the corresponding machine from the store.

The inspection data is visualized inside a web app that uses a responsive design. By leveraging the semantic annotations in the payload this visualization app can also associate the different input data from the inspection machines with the aggregated temperature data from MOSCITO. Both SIWIAT and MOSCITO do not need to rely on a common predefined data model for exploring, visualizing or processing data. Therefore, the integration of shop floor data into business processes can be performed using semantics-driven. These apps may even proxy the interaction to enterprise resource planning (ERP) or manufacturing execution system (MES) systems.

Das Publizierte Paper über den Use-Case ist in der ACM digital library zu finden, eine kostenfreie Variante kann man über den Authorizer-Link der Author-Website beziehen.

5.2 Ondics GmbH

KAPITEL 6

ScaleIT App Design-Prinzipien

In diesem Kapitel werden Design-Prinzipien vorgestellt die dazu dienen eine saubere Implementierung der Software unterstützen. Diese Entwicklungsprinzipien sind eine Sammlung von Best-Practices aus der modernen Software-Entwicklung und fokussieren sich auf die Bedürfnisse des betrieblichen Hallenbodens.

Eine App ist ein logische Konstrukt, dass der verbesserung der User-Experience (UX) dient.

Die konkrete softwaretechnische Architektur der ScaleIT Plattform wird in Kapitel **'ScaleIT Architektur'_** beschrieben.

Die App Compliance Level werden im Kapitel **'Compliance Level'_** beschrieben.

6.1 App Software-Design-Prinzipien

ScaleIT Apps are a model of the pattern of multiple cooperating processes which form a cohesive unit of service. They simplify application deployment and management by providing a higher-level abstraction than the set of their constituent applications. Apps serve as unit of deployment, horizontal scaling, and replication. Colocation (co-scheduling), shared fate (e.g. termination), coordinated replication, resource sharing, and dependency management should be handled automatically for containers in a app.

The applications in an App all use the same network namespace (same IP and port space), and can thus "find" each other and communicate using localhost. Because of this, applications in an App must coordinate their usage of ports. Each App has an IP address in a flat shared networking space that has full communication with other physical computers and Apps across the network. (Adapted from Kubernetes Pods).

6.1.1 Leveraging Bounded Contexts

Zum Zweck der Modularisierung von Software eignet sich das Prinzip des Bounded Context *[DDDEvans] [FowlerBoundedContext]*.

Bounded Context is a central pattern in Domain-Driven Design1. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships. DDD is about designing software based on models

of the underlying domain. As you try to model a larger domain however, it gets progressively harder to build a single unified model. Different groups of people will use subtly different vocabularies in different parts of a large organization. So instead DDD divides up a large system into Bounded Contexts, each of which can have a unified model - essentially a way of structuring the domain into multiple canonical models *[FowlerMultipleCanonicalModels]*.

Bounded Contexts use the principle of Shared- and Hidden Models *[NewmanMicroservices]*, where each Bounded Context exposes an explicit interface to the outside world, the Shared Model, while the Hidden Models are irrelevant to other Bounded Contexts and therefore kept encapsulated privately.

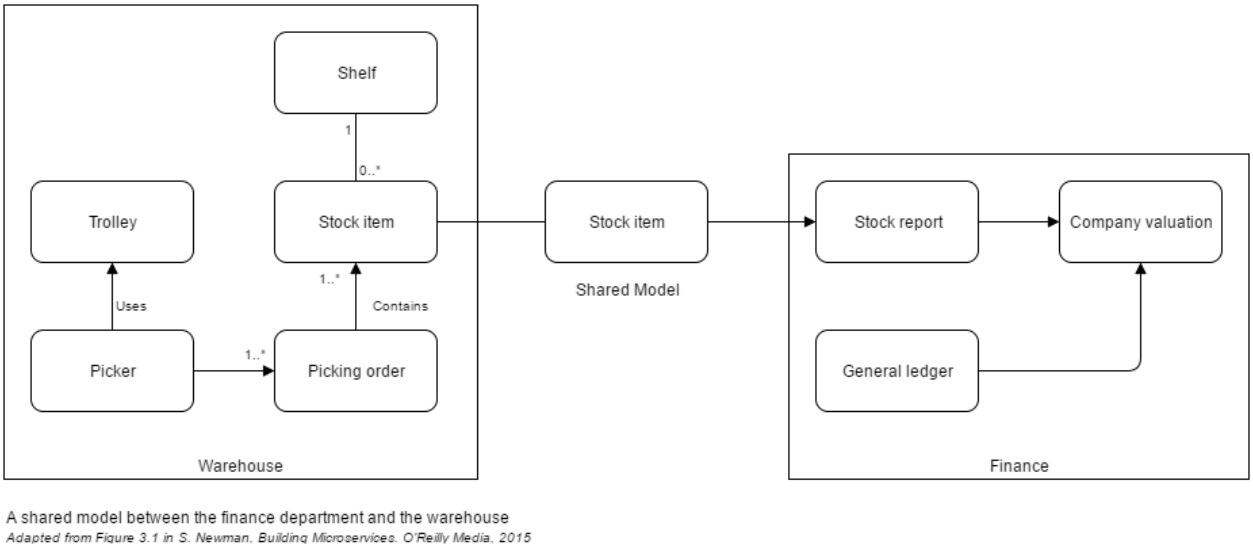


Abb. 1: Shared Model showing two bounded contexts

A Bounded Context is a unit of function with business value, that has high cohesion and reprensents a part of the domain (subdomain) or in other words it is a specific responsibility enforced by explicit boundaries *[BoundedContextExplained]*. Examples: Billing, Machine Control Interface

---

Zu tun: App Richtlinien aus der Wiki hinzufügen

6.2 App User Interface Design

UX Principles

6.2.1 Nielsen's 10 Heuristics for Good UX

These 10 Heuristics help building user friendlier apps and identifying usability issues *[Nielsen10]*.

Visibility of system status The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom Users often choose system functions by mistake and will need a clearly marked „emergency exit“ to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Recognition rather than recall Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Aesthetic and minimalist design Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Help users recognize, diagnose, and recover from errors Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Bemerkung:

Useful Tools: UX Check, http://www.uxcheck.co

Treejack, https://www.optimalworkshop.com/treejack

6.2.2 App Examples

6.3 Bibliography

6 “Microservices Resource Guide @ martinfowler.com.” [Online]. Available: http://martinfowler.com/microservices/.

7 C. van den Thillart, G. Vermaas, M. van der Linden, and J. Vermeir, “microservices-principles @ blog.xebia.com.” [Online]. Available: http://blog. xebia.com/tag/microservices-principles/.

9 “streamprocessing @ cloud.google.com.” [Online]. Available: https://cloud.google.com/solutions/architecture/streamprocessing.

10 “Unix_philosophy @ en.wikipedia.org.” [Online]. Available: https://en.wikipedia.org/wiki/Unix_philosophy#Eric_Raymond.E2.80.99s_17_Unix_Rules.

11 A. Wiggins, The Twelve-Factor App. 2012.

12 “Software Architecture @ Msdn.Microsoft.Com.” [Online]. Available: https://msdn.microsoft.com/en-us/library/ee658124.aspx.

14 The 6 Traits of Reactive Microservices: Isolated, Asynchronous, Autonomous and more [Online]. Available: https://www.lightbend.com/blog/the-6-traits-of-reactive-microservices-isolated-asynchronous-autonomous

Abb. 2: ScaleIT App created in cooperation with SICK AG with a responsive Web interface and meta-data for smartphone homescreen embedding.

15 Richardson Maturity Model (REST), http://martinfowler.com/articles/richardsonMaturityModel.html

16 Rubber Duck Debugging, https://en.wikipedia.org/wiki/Rubber_duck_debugging

KAPITEL 7

ScaleIT Architektur

Don't forget to look at the *App Design Richtlinien* to build a better understanding of what a good App is.

App Readiness Checklisten hier

7.1 Ebenen der Architektur

The ScaleIT architecture allows teams or organisations to manage their own IoT landscape and innovate in their own pace using a parallel IT landscape. This landscape is usually called a shadow IT. This niche for the shadow IT arises from the dichotomous relation between shop floor software engineers and the company wide IT. While shop floor software engineers are enabled to have full control of their deployment environment, the company IT hosts these virtual environments isolated inside the company network. A shadow IT allows the shop floor teams to move independently to the central IT operations.

In this layered architecture, layers below provide software side control for the layers above (bottom up). The top down interaction is facilitated mainly by using pre-built platform services as sidecars. These sidecars are deployed bundled with the Apps (Fig. 3). The figure below illustrates the abstracted planes of responsibility, each housing a different set of Apps:

Domain-Apps Apps that implement business cases reside on the domain App plane. It is the abstraction level at which business case applications are deployed (visualisation Apps or HMIs, machine learning Apps etc.). This is the plane where the process managers (Prozessleiter) would pick click-to-install Apps from an App store in support of the business processes. Workers come in contact with theses apps in their day to day work. Shop floor IT engineers deploy in-house App on this plane. Shop floor software innovation can thus be realized by adding a new or updated set of Apps.

Business-Essentials The business essentials plane houses Apps that are rele- vant to the stability of the domain apps and therefore the business. This is also the plane where the IT team of the company would maintain their backup solutions, audit logs

Platform-Essentials The platform essentials plane houses Apps that are re- quired to run the shop floor eco-system at an App level (App registry, App monitoring, routing, logging, App traces). This is the plane where a business

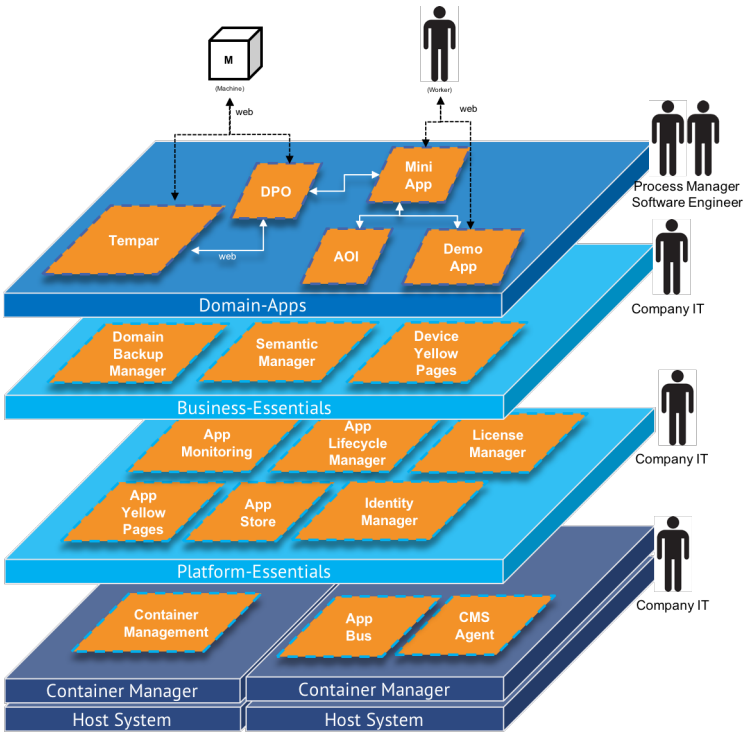


Abb. 1: ScaleIT Architektur mit den unterschiedlichen Ebenen und verantwortlichen Rollen.

would not need to do software lifecycle specific tasks as it will be comprised as open source, open platform distribution. Adapting the Apps on this plane is done through configuration via Web interfaces.

Container Manager The container manager is responsible for the App lifecycle (start, stop, restart, update, replication) and provides an overview of the shop floor eco-system at a container level.

Host System The host system is mainly a operating system running on hardware that is sufficient for the Apps deployed on it. The host system only requires a runtime engine for a con- tainer system (e.g. Docker). All subsequent software is in- stalled henceforth (bottom to top) as containers. Because all software runs inside containers, the host system can be ei- ther physically co-located with the shop floor or somewhere within the company or public cloud. Furthermore, addi- tional host systems may be Raspberry Pi-like devices where only an agent of the container manager is run.

By employing Web-based microservices, continuous deliv- ery, containerization, infrastructure as code and a DevOps culture, we believe that manufacturers can start building IoT solutions and scale them easily with their evolving businesses, especially leveraging the ubiquity of modern Web technologies.

Given an open source reference implementation of the Platform-Essentials plane and corresponding side cars, a manufacturing company can spin up an entire software eco- system and deploy basic applications on it, even when staff capacity is limited. The use case implementation section details a scenario where a manufacturer managed to bring their new application into production in a faster and safer manner.

7.2 Shopfloor Roles Supported by ScaleIT

Viewing from the manufacturing perspective, there are several key roles held by the actors within an Industrial IoT system. Each of these actors is working with different levels of domain abstraction:

Workers Despite a high automation degree, workers are central to the execution of shop floor processes. They need to receive information about the relevant states of the system and use apps as aid in their tasks.

Process manager The design and controlling of shop floor processes lies in the responsibility of the process manager. This actor will usually search for productivity improvements while ensuring that the quality standards are met. By working data driven, the process manager is on the lookout for new applications that can bring benefits to the manufacturing process (dashboards, worker information applications).

Shop floor IT Engineers Employees with software engineering experience, usually in small teams, are tasked with keeping the automation technology or the machine software running. They use their limited capacity to bring innovation through domain specific software onto the shop floor (usually triggered by the process manager).

Company IT The Company IT is responsible for the security and reliability of the company networks and company wide software distributions (Office suite, CAX engineering software). In contrast to the other actors, the company IT very seldomly comes into contact with shop floor applications.

Independent vendors Vendors are 3rd party software distributors and software component providers across all actors. Companies usually incorporate 3rd party software for the entire automation pyramid. However, most of the time, system integration is done manually in an ad-hoc manner by employees.

The above roles are a generalisation and the responsibilities may be be split between several persons. However, these roles are central to the continuous functioning and evolution of the shop floor.

7.3 App Anatomy

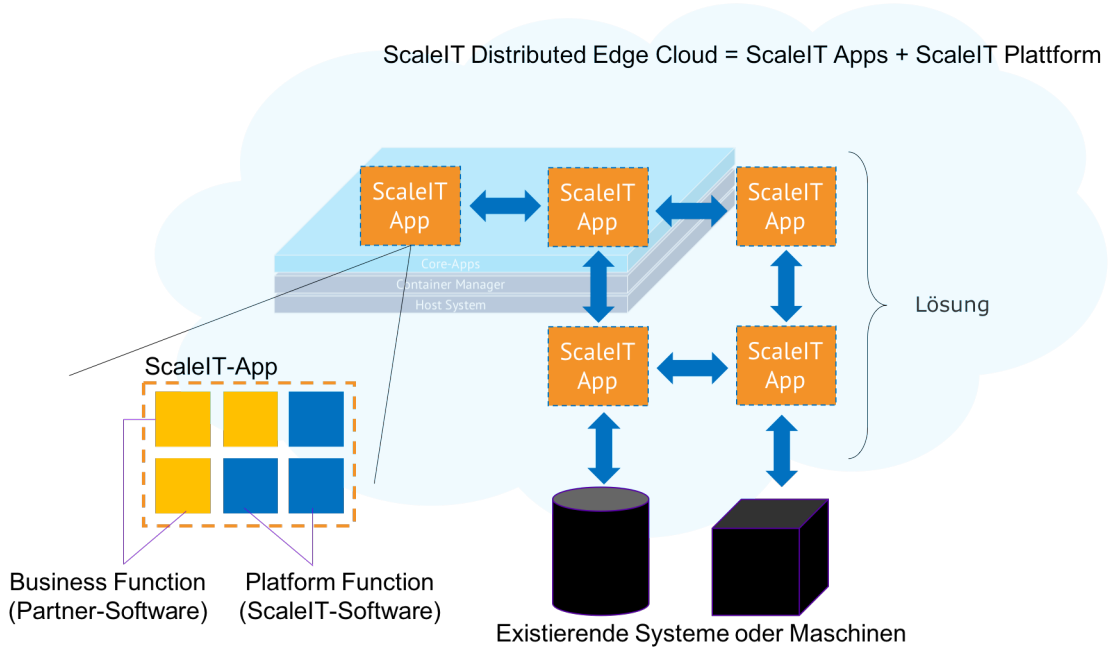


Abb. 2: Die Umgebung einer App ist der Shop Floor und sie haust in der ScaleIT Distributed Edge Cloud.

7.3.1 Sidecar Pattern

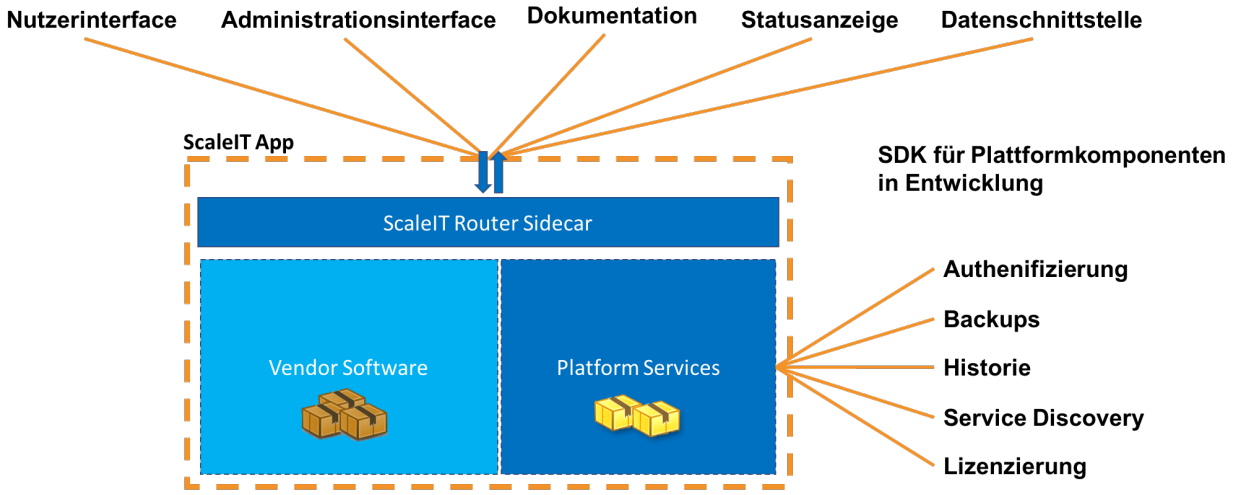


Abb. 3: Eine ScaleIT-App besteht aus der Nutz-Software, die einer direkten Werstschöpfung dient, sowie der Plattformkomponenten die über das Sidecar-Pattern eingebunden werden.

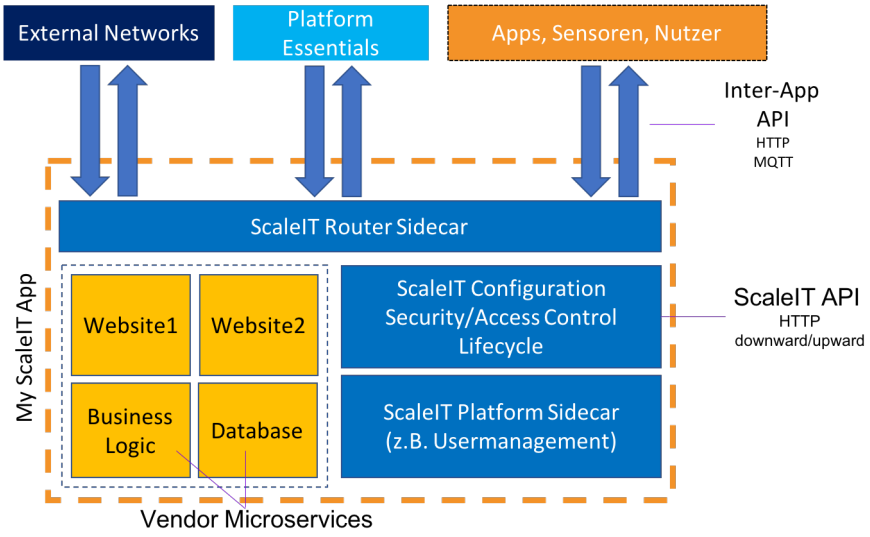


Abb. 4: Nach dem Prinzip des 1-Prozess/Container, wird die Software innerhalb eine ScaleIT App in mehrere unterschiedliche Container verpackt, die gemeinsam die gesamtfunktionalität abdecken.

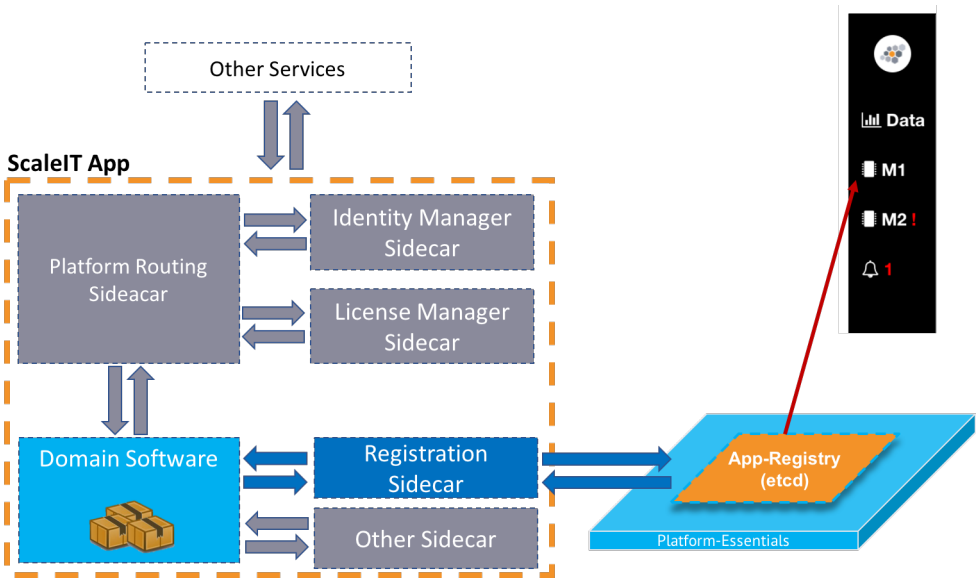


Abb. 5: Durch das Sidecar-Pattern werden Plattformfunktionalitäten hinzugefügt.

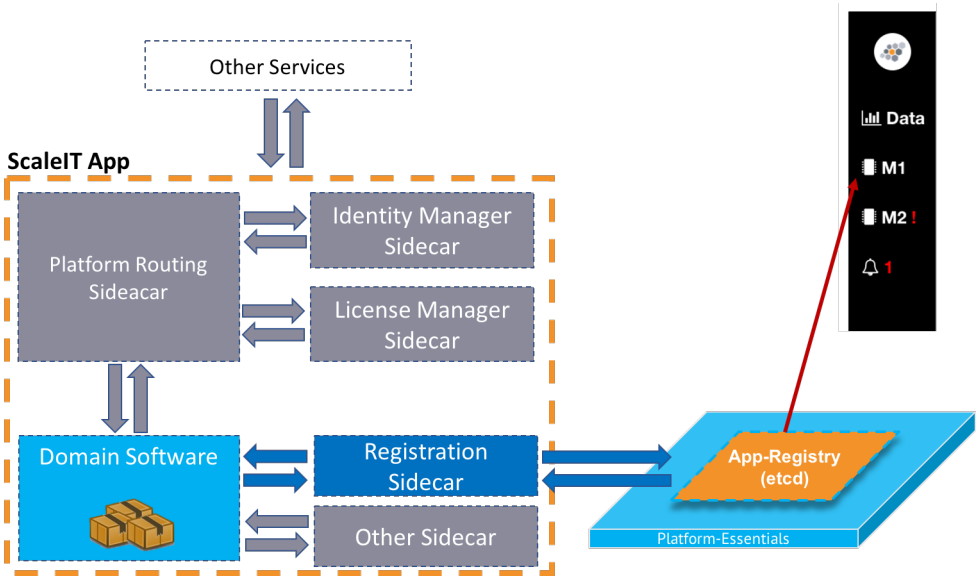


Abb. 6: Caption Text

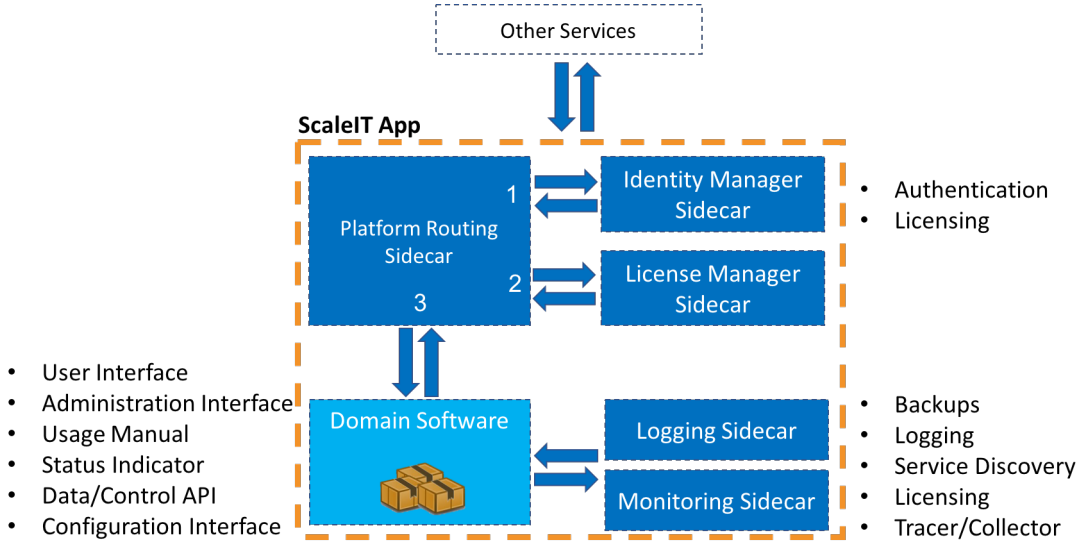


Abb. 7: Caption Text

KAPITEL 8

Industry 4.0 App Readiness

Docker Compose ist die Technologie der Wahl, wenn es um die Aggregation der Bestandteile einer App geht. Da Docker Compose nur ein Werkzeug ist und nicht ein elementarer Baustein der Plattform kann es durch z.B. Kubernetes Helm Charts oder Rancherfiles ersetzt werden, ohne die Struktur der App zu beeinflussen.

Zu tun: Work in progress

8.1 App Containers (Docker Subsystem)

☐ Das Git-Repository nutzt die ScaleIT-Ordnerstruktur (Domain Apps, Sidecars Ordner)

```
<app_name>
├── docker-compose.yml
├── dc.build.yml [optional]
├── .env.default
├── .env.test.default [optional]
├── .env.staging.default [optional]
├── .env.production.default [optional]
├── README
├── Resources/
| ├── Documentation/
| ├── Rancher/
| | ├── catalogIcon-<app1>.svg //jpg,png or other format ok
| | ├── appHubIcon-<app1>.svg //jpg,png or other format ok
| | ├── config.yml // meta-data about the app
| | └── Screenshots/ // App Screenshots
| | | └── 1.jpg
| | └── Versions/
| | .   └── 0/
| | .   . └── docker-compose.yml // compose version 2 due to compatibility reasons
| | .   . └── rancher-compose.yml
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
| └── Documentation/
| └── README
├── Domain Software/
| ├── DomainContainer1/
| | ├── Dockerfile
| | └── <some other files>
| └── DomainContainer2/
| └── Dockerfile
└── Platform Sidecars/ [optional]
. ├── SidecarContainer1/
. | ├── Dockerfile
. | └── <some other files>
. └── SidecarContainer2/
. . └── Dockerfile
```

☐ Dockerfile -> Docker Build -> Image: includes all needed Dependencies (Self Contained-ness/in sich geschlossen)

☐ Docker Compose declares all needed Services (no other services will be started within the app)

☐ The App can be started with a single „click“ (docker-compose up)

☐ Docker Compose declares all needed Volumes (Data Volume + Log Volume)

☐ The App can be stopped and restarted without domain data loss or (docker-compose stop/restart)

☐ The App containers can be deleted without domain data loss (docker-compose down)

☐ The App containers can be replaced by new containers without domain data loss or corruption (docker-compose down + build + up)

☐ The timezone set for the container is UTC. See *Time Zone Details, Why UTC?*

☐ Data Migration check may be necessary

☐ The created containers shut down properly (no PID 1 zombies)

☐ Adhere to the Dockerfile best practices

8.2 App Interfaces

Sinn dieser Interfaces: „Eine Web-UI zu haben um Administration und Datensicht auf die App und das was sie macht zu erlauben“.

Administration Endpoint */admin*

- admin/config
- admin/doc
- admin/log
- admin/status

User Endpoint */user*

- user/doc
- user/status

Developer Endpoint */dev*

- dev/doc

- dev/rest
- dev/swagger.yaml

Zu tun: Insert Link to Spec as Swagger file

8.3 App Catalog Entry

☐ A separate git repository contains the meta-data from the Resources/Store directory in a Rancher-compatible directory structure

☐ Auto-generated entries for this repository (e.g. git post commit hooks that push meta-data to this app-store repository)

```
-- templates
|-- <app1>
| |-- 0 // App1-Version 0
| | |-- docker-compose.yml
| | |-- rancher-compose.yml
| | |-- answers.txt //environment variables for rancher-compose
| | |-- README.md
| |-- 1 // App-Version 1
| | |-- docker-compose.yml
| | |-- rancher-compose.yml
| | |-- README.md
| |-- catalogIcon-<app1>.svg //jpg or other format ok
| |-- appHubIcon-<app1>.svg //jpg or other format ok
| |-- config.yml // meta-data about the app
| |-- README.md
|-- <app2>
| |-- 0 // App2-Version 0
...
```

Contents of the *config.yml*

```
name: # Name of the Catalog Entry
description: |
# Description of the Catalog Entry
version: # Version of the Catalog to be used
category: # Category to be used for searching catalog entries
maintainer: # The maintainer of the catalog entry
license: # The license
projectURL: # A URL related to the catalog entry
```

This information is strongly inspired by the Rancher Catalog system: [http://rancher.com/docs/rancher/v1.2/en/catalog/private-catalog/{]}(http://rancher.com/docs/rancher/v1.2/en/catalog/private-catalog/)

A catalog entry generator can be found here: [https://github.com/slashgear/generator-rancher-catalog{]}(https://github.com/slashgear/generator-rancher-catalog)

8.4 App Documentation

☐ Readme states the purpose of the App

☐ Readme lists the services and describes them shortly

☐ Playbook includes App Lifecycle commands (pull, start, stop, upgrade)

☐ FAQ

☐ Known common Errors

☐ Architecture Diagramm (eg. UML Deployment Diagramm)

☐ Readme includes logo and screenshots

☐ App Requirements (RAM, CPU, HDD)

☐ Examples:

- Chronocommand
- ScaleIT Gitlab

ScaleIT App Compliance Level

☐ App has a User UI

☐ App has an Administration UI

☐ App has the networking information included (routing address)

8.5 App Behaviour

☐ Logging

☐ Graceful degradation

8.6 Software Engineering

☐ Reactive Design (App Richtlinien)

☐ https://projects.teco.edu/projects/scaleit-ap2/wiki/Richtlinien_App-Entwicklung

8.7 Development Process

☐ Automated build pipeline

☐ Continuous Integration

☐ Use Dynamic Port ranges 49,152 through 65,535.

8.8 Time Zone Details, Why UTC?

Why Not {PST, GMT, PDT, etc}?[#serverutc]_

- UTC has no Daylight Savings
- Uniform time across all sites, factories and offices

- Decreases data corruption chances due to inconsistent time zones
- Standardized time across all our Apps will ensure logs, databases and all components relying on the time will function in a predictibale and interoperable way.

Bemerkung:

This will move the problem up into the UI layer. We recommend working with UTC inside the App logic and converting UTC to

Tech Tip: Using the angular DatePipe in the UI will help you achieve this easily[3]. Look in the programming language of your choice to find similar useful features.

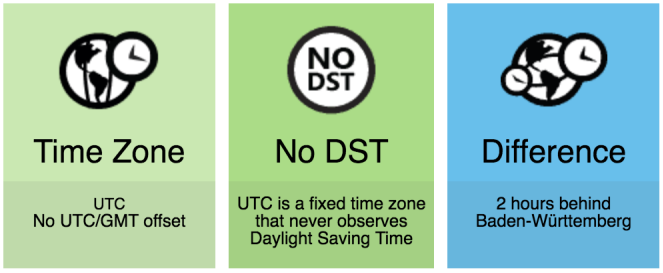


Abb. 1: UTC in a glance[1]

[3] Angular DatePipe, https://angular.io/api/common/DatePipe
[1] UTC in a glance, https://www.timeanddate.com/worldclock/timezone/utc

KAPITEL 9

Networking

9.1 Port-Vergabe

9.1.1 Network Ports für Apps

Als Konvention werden Ports innerhalb der Plattform zwischen den dynamischen Ports 49,152 und 65,535 vergeben.

- Port range 49500 through 49599 for platform components
- Port range 49600 through 49699 for business essentials components
- Port range 49700 and above for apps

Port ranges are assigned for companies, e.g. 49700-49900 for Ondics Apps

Tabelle der Port-Konventionen in ScaleIT:

Port	Komponente	Organisation
80	ScaleIT Proxy	ScaleIT-Org
49500-50000	**ScaleIT-Org**	
49500	Rancher Server	
49501	App Registry	
49502	Identity Management	
49503	MQTT Broker	
ScaleIT Entwicklungs-Partner		
50000-50500	ATOS	
50500-51000	Fraunhofer	
51000-51500	KIT	
51500-52000	ondics	
52000-52500	SICK	
53000-53500	SmartHMI	
54000-54500	ZEISS	
ScaleIT Anwendungs-Partner		
54500-54600	digiraster	
54600-54700	Feinmetall	
54700-54800	microTEC Südwest	
54800-54900	Rood Microtec	
59000-55000	Universität Stuttgart	
Externe Partner		
55000-55100	ITstrategen	
55100-55200	YUMA technologies	
55200-55300	Ingenieurbüro Teichgräber	
55300-55400	HS Esslingen	
55400-65536	Frei	

9.2 Funktionen der Plattform Essentials (Zentrale Services)

9.2.1 App Registry

In der Referenzplattform durch ETCD implementiert.

- kann von jeder app als key-value store genutzt werden
- der top-node muss der fqdn-name der app sein (z.b.

  /pacman_1.host_1.scaleit)

weitere regeln dazu: - jede app sollte nur die einträge der eigenen app lesen - die zentralen apps haben spezielle root-nodes, z.b.

  /apphub

- wenn eine app gelöscht wird kann der node gelöscht werden

  (z.b. von einem registry-cleaner-prozess)

- die apps dürfen den etcd nicht als massendatenspeicher

  nutzen

- systemweite einstellungen finden sich unter /scaleit,

  z.b. /scaleit/language oder /scaleit/brand/company-name... alle apps solten das berücksichtigen

9.3 Fragen zu den Anforderungen and das Networking in ScaleIT

- Der Zugang nach außen soll möglich sein (z.B. „ping google.de“)
- **Wie werden zentrale Services aus einem Container angesprochen?**
  - Ideal wäre sowas wie „ping apphub.coreservice.scaleit“?
- **Wie werden andere Container angesprochen? Achtung: Apps können mehrfach instantiiert werden oder auf verschiedene**
  - Ideal wäre „ping pacman_1.host_1.scaleit“
- **Wie können Services von außen angesprochen werden?**
  - Hierzu sollte im Firmennetz *.scaleit.company.com an die ScaleIT-Plattform weitergeleitet werden
  - Der DNS-Name pacman_1.host_1.scaleit.company.com würde dann zur App führen

Was muss in den docker-compose stehen? Springe zu *Docker Compose network configuration*

Was muss auf Docker-Ebene passieren (sollen wir ein eigenes Docker-Netzwerk definieren?) Ja, auf Docker (Container) Ebene wird es mehrere Netzwerke geben. Siehe *ScaleIT App Networking*

Docker Compose hier!

9.4 ScaleIT App Networking

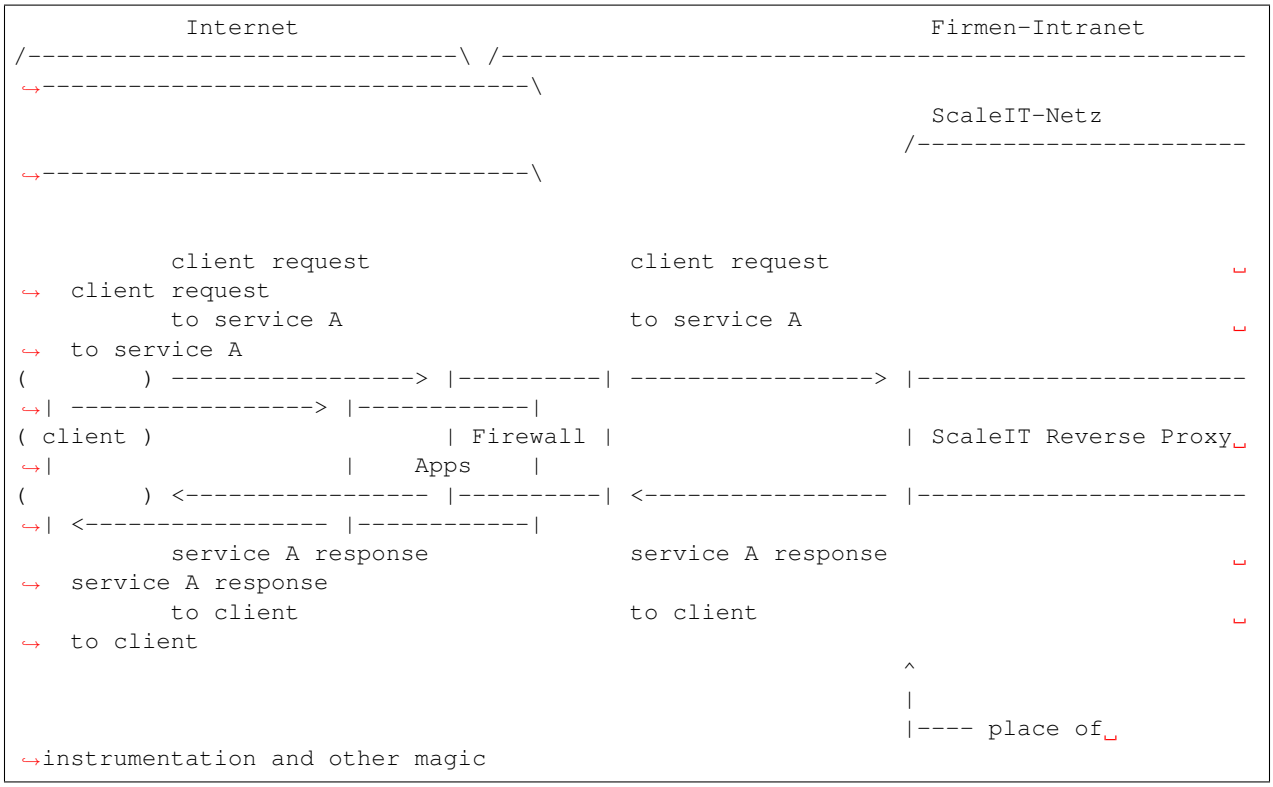
Ein logischer Server mit zentralen Diensten, die nur einmalig im System vorkommen können:

- Rancher

- App-Hub
- Licence Manager
- Yellow Pages (ETCD)
- LDAP / OAuth-Server

Ein oder mehrere Hosts, auf dem Apps laufen.

Diagrams inspired by _1Backend



9.5 HTTP Request Headers

Copy from Github repo

KAPITEL 10

Kommunikation auf der ScaleIT Plattform

Die Kommunikation auf der ScaleIT Plattform

Es wird unterschieden zwischen

Meldungen die an eine einzige Entität gehen

an eine einzige Instanz oder

10.1 Kommunikation zwischen Apps (App-2-App)

10.1.1 1:1 Kommunikation (Peer2Peer,Resourcenanfragen)

10.1.2 1:n Kommunikation 9

10.2 Kommunikation zwischen Apps und der Plattform (App-2-Platform)

Die Kommunikation zwischen Apps und der Plattform (App-2-Platform) wird über den Sidecar-Mechanismus realisiert. Entwickler binden die Platform-Sidecars ein und nutzen die standardisierten HTTP APIs, um darauf zuzugreifen. Dabei ist zu beachten, dass die Sidecars im internen Netzwerk der App mit ihren docker-compose Service Namen angesprochen werden können.

Requests die durch das Routing Sidecar an die App gegeben werden enthalten je nach ausbaustufe der Plattform Informationen zu Identität, Authorisierung, Lizenz etc. - z.B. als HTTP Header. Dadurch, dass der Entwickler dem Sidecar vertraut können diese in der App genutzt werden ohne die zentralen Instanzen anzusprechen. Dadurch wird die Abhängigkeit zu einer bestimmten Plattforminstanz reduziert.

Die Sidecars melden den zentralen Plattformkomponenten (Platform-Essentials?) und Business Essentials wenn es Änderungen gibt. Z.B. HTTP API.

Weil sich hinter diesen zentralisierten Komponenten meist ein Cluster von Anwendungen verbirgt, … Master-Komponenten und Slave Komponenten…

10.2.1 Kommunikation mit der App Registry

10.2.2 Kommunikation mit dem Identity-Manager

10.2.3 Kommunikation mit dem License-Manager

KAPITEL 11

ScaleIT Lifecycle Section

The lifecycles of different aspects of the ScaleIT ecosystem are presented in this chapter.

11.1 From Idea to App

11.2 App Store Lifecycle

- Review Process
- etc.

11.3 Deploy Lifecycle

also look into googling: circle ci lifecycle

The stages through which an App goes until it reaches

- The pull request runs tests and is marked as good to merge into master
- The engineer merges the PR and tests are again run against master
- That commit of master deploys to our centralized staging server
- Sanity checks run automatically on the staging server
- That commit then deploys to production
- Sanity checks run on production
- Success and failure notifications are sent via Slack at every step
- Rollback possible for several weeks (depending on the storage settings)

Warnung: Don't forget to create release notes and bump the necessary versions.

In the event of a rollback, an engineer must manually deploy from their machine to cause the latest release status to change and implicitly "re-activate" continuous deployment.

There should be checks with kill switches along the way (e.g. for every environment and one final manual approval from shop floor process owners).

Important people (on call) should get notified of deployments statuses

KAPITEL 12

Sicherheit (Security) auf der ScaleIT Plattform

12.1 Für Entwickler

Gegen wen und gegen welche Angriffe schützen wir unsere ScaleIT Instanz?

Externe Angreifer: Im Falle einer horizontalen Vernetzung oder Erreichbarkeit der Infrastruktur aus dem Internet oder durch ungesicherte WLAN Access-Points.

Interne Angreifer: Interne Angreifer sind immer ein hohes Risiko. ScaleIT bietet durch den App-Aufbau aber Mechanismen, die den Zugriff limitieren, wie z.B. separate Netzwerke für sensible Komponenten innerhalb eine App oder Rollenbasierte API-Zugriffskontrolle.

Schlechte Software: Eine Software kann wissentlich oder unwissentlich Schaden verursachen. Ein Beispiel dafür ist die Ressourcennutzung (PIDs, CPU etc.) oder der Zugriff auf das root Filesystem. Einstellungen die die Angriffsfläche mindern sind in dem ScaleIT SDK eingebaut. Es wird den Entwicklern empfohlen die Standardwerte zu nutzen. Der Review-Prozess im offiziellen ScaleIT-App-Store verifiziert (automatisch) jede neue App, um solche Szenarien zu vermeiden.

Zu tun:

Welche innerhalb und außerhalb von Docker liegenden sicherheitsrisiken gibt es und was tun wir dagegen?
Beantworten

Wenn das „ScaleIT System“ nicht auf Kommandozeilenebene zugänglich ist, fallen damit wesentliche risiken weg?
Nein, solange Container im privilegiertem Modus, mit dem Docker Socket arbeiten oder auf das root Filesystem Zugriff haben, kann man das nicht pauschal sagen.

Zu tun:

Wie organisieren wir das ssl-zertifikat management?

Brauchen wir eine eigene certificate authority, um an die apps zertifikate auszustellen bzw. diese zu überprüfen?
Beantworten

Wie gehen wir mit ablaufenden zertifikaten um? Beantworten

Zu tun:

Welchen Bezug gibt es zu den standards, die auf dem hallenboden und im betrieblichen umfeld gelten (bsi, . . .)?
Beantworten

KAPITEL 13

TEST

RST Tests here

KAPITEL 14

Indices and tables

- genindex
- modindex
- search

Literaturverzeichnis

[DDDEvans] 5. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[FowlerMicroservices] 10. Lewis and M. Fowler, "Microservices @ Martinfowler.Com," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html.

[NewmanMicroservices] 19. Newman, Building Microservices. O'Reilly Media, 2015.

[FowlerBoundedContext] "BoundedContext @ martinfowler.com." [Online]. Available: http://martinfowler.com/bliki/BoundedContext.html.

[FowlerMicroservicePrerequisites] "MicroservicePrerequisites @ martinfowler.com." [Online]. Available: http://martinfowler.com/bliki/MicroservicePrerequisites.html.

[FowlerMultipleCanonicalModels] 8 "MultipleCanonicalModels @ martinfowler.com." [Online]. Available: http://martinfowler.com/bliki/MultipleCanonicalModels.html.

[BoundedContextExplained] "DDD - The Bounded Context Explained." [Online]. Available: http://blog.sapiensworks.com/post/2012/04/17/DDD-The-Bounded-Context-Explained.aspx.

[Nielsen10] Jakob Nielsen, Ten Usability Heuristics, January 1, 1995, online: https://www.nngroup.com/articles/ten-usability-heuristics/