Progress in Computing Applications

DLINE JOURNALS

Print ISSN: 2278 – 6465
Online ISSN: 2278 – 6473

PCA 2025: 14 (2)
https://doi.org/10.6025/pca/2025/14/2/90-98

Agile Requirement Engineering for Scalable Cloud Systems in Automated Vehicle Ecosystems

Armin Mokhtarian
Informatik 11 – Embedded Software
RWTH Aachen University. Germany

Alexandru Kampmann
Informatik 11 – Embedded Software
RWTH Aachen University. Germany

Bassam Alrifaee
Informatik 11 – Embedded Software
RWTH Aachen University. Germany

Stefan Kowalewski
Informatik 11 – Embedded Software
RWTH Aachen University. Germany

Bastian Lampe
Institute for Automotive Engineering
RWTH Aachen University. Germany

Lutz Eckstein
Institute for Automotive Engineering
RWTH Aachen University. Germany

ABSTRACT

The UNICARagil project aims to develop a cloud-based infrastructure supporting four types of fully automated vehicles—autoELF, autoTAXI, autoCARGO, and autoSHUTTLE—each with distinct functional requirements. Given the project's scale, involving over 100 developers from 15 academic chairs and 6 industrial partners, agile requirement engineering becomes crucial. This paper introduces a structured methodology combining agile and classical requirement engineering to accommodate evolving requirements across varied use cases. A template-based survey was designed to gather 47 use cases, which were transformed into 42 system requirements using a pattern-based approach that emphasizes completeness, clarity, and traceability. These were then translated into system design components and cloud services, resulting in over 70 API endpoints.

The design leveraged OpenAPI for endpoint specification, enabling automated code generation using tools like the OpenAPI Generator and bootprint to produce Flask server stubs and documentation. A GitLab CI pipeline ensures continuous integration by generating code and documentation upon each update. This iterative, modular, and scalable process allows rapid prototyping and adaptability, aligning well with the project's dynamic development environment. The proposed methodology proves effective in managing complexity and supporting the agile development of a cloud system for highly automated and networked vehicles.

Keywords: Automated Vehicle Ecosystems, Agile Requirements, Cloud Systems, OpenAI

Received: 4 February 2025, Revised 29 March 2025, Accepted 6 May 2025

1. Introduction

Depending on the particular use case, autonomous vehicles enter larger systems. In the case of an automated shuttle, a coordinating group needs to command its larger fleet of vehicles while providing humans with the ability to summon a car through their smart phone. And parcel delivery, yet another use case pursued by numerous initiatives, also requires a coordinating party. As shown in Fig. 1, these coordinating parties can be implemented as cloud services and are therefore a crucial component of the automated vehicle ecosystem. Specifically, they provide the infrastructure through which communication takes place within the overall system. The different actors could either store or request data and coordinate with one another through the cloud system. Hence, the cloud system provides the primary interfaces for data exchange among actors: a user requests a taxi via a mobile app, or an automated delivery vehicle requests its next destination. Besides being the key player in communication, the cloud system also aids the entire ecosystem of automated vehicles by aggregating, fusing, and processing data.

The development of an ecosystem centered around automated driving is pursued in the UNICA Ragil project, which is a multi-partner project aiming to develop four fully automated vehicles of different characteristics [14]. As the vehicle on-board software is implemented following a service-oriented software architecture (SOA) [5], the cloud software also follows a SOA approach. A total of over 100 developers spread over 15 university chairs and 6 industrial partners are involved in this four-year project. Having a large number of participants, the communication and coordination of the development becomes extra challenging. First, because researchers and developers are spread among different locations. Second, because they have different academic backgrounds. This may result in numerous, non-uniform and poorly communicated specifications. Additionally, our cloud system has to serve four vehicle types which all share the same platform concept but have a different use.

- autoELF: A family vehicle intended for private use [12].
- autoTAXI: A vehicle for short-term general use.
- autoCARGO: A fully automated parcel service.
- autoSHUTTLE: Public transportation of multiple people.

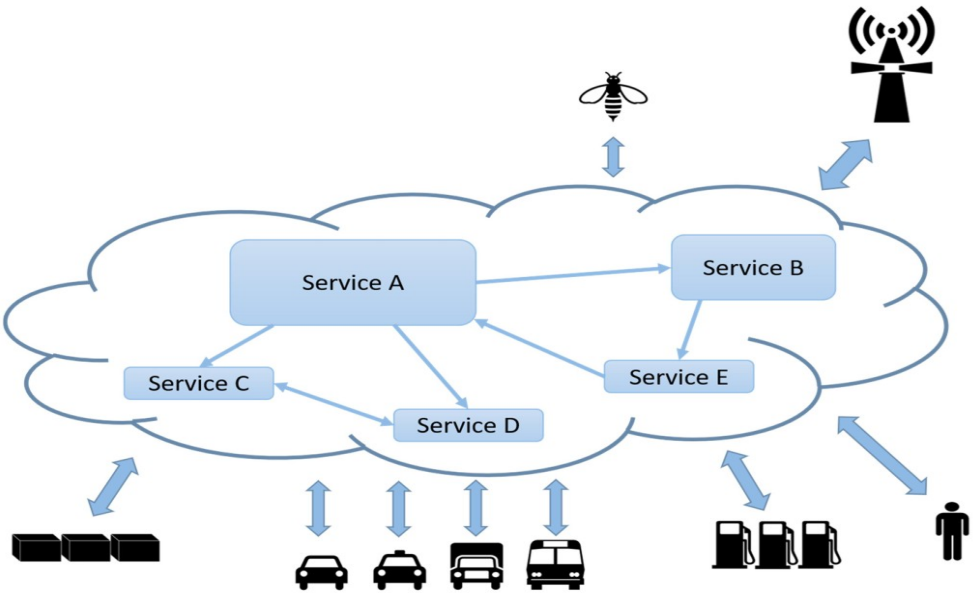

Figure 1. An overview of the cloud and its actors in the UNICARagil project The ecosystem of UNICARagil includes four different vehicle types, a charging as well as a packing station, a control room and an info-bee (drone) [14]

The vehicle variants rely on a cloud system for coordination and for optional input to the automation algorithms through an external environment model. Although there are several commonalities between these vehicle types, they have different requirements for the cloud. The auto CARGO, for example, is the only platform that needs additional interfaces to communicate with a logistics management service. Hence, we have platform specific requirements in addition to requirements arising from automated driving scenarios. Furthermore, due to the early development stage, different platforms and a large number of developers with different backgrounds, the process from requirement engineering to implementation becomes very challenging. In this case, agile development is unavoidable. In particular, since the requirements change during the implementation, we need to simplify the overhead for adjustments.

Another challenging aspect arises since the cloud system and the project are developed simultaneously. In the early stages of a project, the requirements and specifications of the final system are often unavailable. Thus, the simultaneous development of the cloud and the overall system requires a scalable and adaptable system for being able to adjust to changing requirements. Consequently, this requires a modular, expandable and adjustable concept. The necessity of an adaptable cloud system increases with an increasing number of stakeholders. In this paper, we present our approach for the design of a cloud system to overcome the aforementioned challenges.

1.1 Related Work

Cloud systems have various use in the automotive domain. [7] claims, that cloud systems are the go-to solution for deploying frameworks suitable for automotive tasks. A cloud-based artificial intelligence framework for continuous training and self-driving is presented by [8]. Their system supports the collection of data which is used to develop and train machine learning models to leverage the cloud as a model performance booster. Hence, the cloud is an essential component in their ecosystem.

According to [3], the most common reason for software project failures is bad or incomplete requirements engineering. Therefore, requirement engineering plays a critical role in our approach. Several methods and guidelines were developed in order to not only prevent project failures but also to allow an efficient approach. A novel approach, which is investigated by Paetsch et al. [9], is to combine classic requirement engineering concepts, e.g. Waterfall model [1], with agile methods like Scrum [13].

2. Method

A system development lifecycle usually consists of the following steps. By starting with a *requirements analysis*, the developers investigate the properties and qualities their system should provide. At this phase, detailed communication with the customer is required to elaborate a solid groundwork. The *system design*, which includes the complete hardware and software setup is derived based on the requirements. Consequently, the system design is broken down into modules in the *architectural design* phase. In the *module phase*, these modules are designed in detail to be implemented afterwards.

If it turns out that the requirements were not levied correctly or changed during the development lifecycle, it may require to restart the development process from scratch in the worst case. Thus, in order to reduce the overhead, it is desired to have robust requirement engineering and an adaptable development process. Agile requirements engineering methods like Scrum are prepared for new or changing requirements as their incremental mode of operation intends to provide adaptability to the current set of tasks. Furthermore, a modular groundwork enables the basis for implementing new requirements.

In this section, we present a methodology, which relies on template-based requirement engineering. This method enables to adapt to new requirements with lower effort and thus provides the basis for an incremental way of working.

2.1 Requirements Engineering

In order to have consistent feedback, we created a survey with concrete questions, which was handed out to all project stakeholders. Moreover, to take care of the difficulty regarding the different academic backgrounds, we asked for use cases instead of operational requirements. To illustrate the interaction of the use cases as well as the actors within the system, UML diagrams can be used [10]. However, the number of attributes and details we want to capture would cause UML diagrams to appear confusing. Furthermore, due to a large number of participants, we could not assume that everyone is familiar with UML. Hence, we decided on a questionnaire with 8 attributes which are listed in Table 1 and adapted from [2]. This assures that the collected use cases all have the same format and enable easy extraction of the essential information. Our survey resulted in 47 use cases that had to be evaluated and processed.

Although we tried to get uniform results by providing a specific template, the survey showed that the participants answered the questionnaires with a varying level of detail. Thus, the next step of our method transforms the questionnaires into a uniform, understandable and transparent representation. For this purpose, as recommended by Christine Rupp [11], we investigated the results concerning completeness, consistency, understandability, necessity, feasibility, clarity and traceability. To assure this quality features, we decided to use a pattern-based approach shown in Fig 2. This approach maps a use case into four parts.

ID	3.5
Use Case Title	Deliver Notification
Description	Customer is notified about the delivery of his parcel.
Actors	Cloud, Customer (App), Parcel Box
Frequency	Daily
Condition	Parcel Box has booked delivered parcel correctly.
Guarantee of Success	Customer can pick up his parcel at the parcel box.
Trigger	Parcel was delivererd to the parcel box by autoCARGO.
Actions	1. Cloud notifies Customer via App about delivered parcel.
	2. Customer enters pin code at parcel box.
	3. Parcel box provides parcel.
	4. Parcel box notifies the cloud about the parcel pickup status.

Table 1. One of 47 use cases gathered with the template-based survey in the UNICARagil project

1. Logical Operator

Most of the functionalities and processes start after a series of preconditions or are triggered through events. Therefore, the temporal component is mapped here.

2. Priority

Different use cases have different priorities for the same functionalities in the cloud. For being able to distinguish between the relevance of a requirement, a three-level rating system is introduced.

3. System

Requirements that are created by this approach are phrased in an active sentence. Since these requirements are gathered in the context of the cloud only, it is always the subject.

4. Process

The main focus of every requirement is the functionalities of the system. At this point, the desired system behavior is described. After transforming 47 use cases into this pattern with regard to quality features, 42 requirements were derived.

2.2 System Design

After processing the gathered use cases into requirements, the next step consists of breaking down the cloud system in to individual cloud services. We will use the exemplary use case *3.5 Deliver Notification* shown in Table 1 to explain the procedure. This use case describes the situation of an auto CARGO delivering a parcel to the parcel box. Afterwards, the user gets a notification via the App and receives a pin code that is needed to pick up the parcel at the parcel box.

In this case, there exist three actors: The user, the parcel box and the cloud. This scenario creates the need for two services to manage user and parcel box data. Hence, the *user management* and the *logistics management* are introduced. Any endpoint can use their API to store and request data. Now, we use the Actions from Table 1 to derive the necessary communication interfaces of the aforementioned cloud services. The resulting relation table is shown in Table 2.

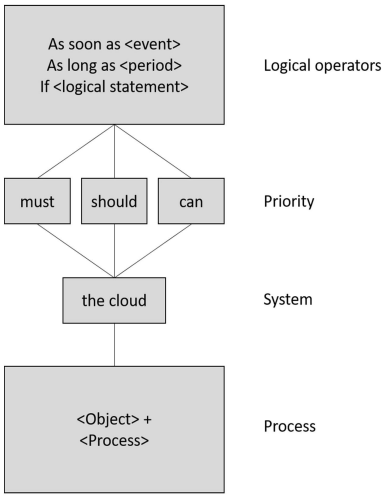

Figure 2. Pattern based approach for requirement description adapted from [11]

3.5 Deliver Notification			
Action	**From**	**To**	**HTTP-API**
3.5.1	Logistics Mgmt	User Mgmt	PUT:/parcel/status
3.5.2	App	User Mgmt	GET:/user/parcel/receive
3.5.3	Parcelbox	Logistics Mgmt	HEAD:/infrastructure/ parcelbox/{PBOX_ID}/ pincodecheck
3.5.4	Parcelbox	Logistics Mgmt	PUT:/infrastructure/parcelbox/ {PBOX_ID}/parcel/{PARCEL_ID}/ status

Table 2. System view on an exemplary use case

In the course of our process, we maintain traceability by linking use case questionaries, derived requirements and final system specifications with an ID. This allows for identifying components that may be affected by a change in the underlying requirements. Besides an ID, the system view contains Action, From, To and API.

For the first Action of this use case (3.5.1), the user management provides two endpoints. One is offered by the cloud internal HTTP API for the logistics management and the other one is offered by consumer mobile application HTTP API. We determined

PUT /parcel/status (1)

to be accessed by the logistics management for notifying the user management about parcel status changes. Further, we determined

GET /user/parcel/receive (2)

to be used by the App for requesting status information about parcels. Its response also provides the pin code for the parcel pickup. The next Action (3.5.2) does not describe any interaction with the cloud and can be skipped. Action 3.5.3, in turn, needs to be handled by

HEAD /infrastructure/parcelbox/pbox_id/pincodecheck (3)

in order to check if the combination of parcel and pincode is correct. Finally, Action 3.5.4 asks the system to update the status of the parcel which is done by

PUT /infrastructure/parcelbox/pbox_id/parcel/parcel_id/status (4)

This approach proved itself to be structured and expedient, as it allowed us to quickly come to an initial system design consisting of more than 20 cloud services with over 70 endpoints. The resulting HTTP-APIs are specified with OpenAPI1 which is an API description format for REST APIs. The specifications are written in YAML and thus are readable to both humans and machines.

The YAML files are processed by the OpenAPI Generator to generate a Python flask server [4]. Furthermore, bootprint2 is used to generate static HTML pages of the OpenAPI specification. We decided for this setup, in order to enable rapid prototyping during our development process

2.3 Code Generation

Since OpenAPI was used to specify the HTTP API for every service, it is possible to convert those into a static HTML page by using boot print. Furthermore, those specifications are used to generate a Python flask server with code stubs for the implementation of all HTTP APIs. This code does not contain the actual functionality of the service but is runnable and allows for speeding up the implementation process. For this purpose, a Git repository with Git Lab Continuous Integration was set up. Tagging a commit with generate_code, generate_doc

[1]https://swagger.io
[2]https://github.com/bootprint/bootprint

or generate when pushing to the master branch, the Git runner is triggered. This generates the HTML documentation, Python flask servers or both for all services. Finally, everything just generated is pushed to the Git repository. Hence, for every change made to the specification, the code is generated automatically and can be tested directly.

3. Conclusion

Based on the Cloud system in the UNICA Ragil project, we presented a methodology to deal with arising challenges in a multi-partner project. Together with a large number of different stakeholders, covering four different platforms makes the development challenging. Thus, our method was chosen to provide consistent and scalable requirements engineering. Furthermore, due to the early stage within the overall project, it was of high importance to use a method which provides fast prototyping and is adaptable to upcoming changes in the requirements

Therefore, we first created a template-based survey that was handed to everybody who intends to work with the cloud. The survey returned 47 use cases which then were examined regarding quality measurements and processed afterward. A pattern-based approach helped to transform the use cases into 42 requirements. Based on these requirements, we derived relation tables to provide a structured way to define interfaces and endpoints. The interfaces were specified with OpenAPI, which enabled the generation of Python flask server code and an HTML documentation. Both code and documentation then are pushed to Git and allows for automated test with GitlabCI[3].

We presented a method that allows processing new requirements into the system with lower effort to allow an incremental way of operation. At the same time, a modular and lightweight groundwork derived by the template-based requirements engineering enabled fast prototyping. This methodology combined classic requirements engineering and agile development. Neither of both would have suited our needs if they were used separately. Our methodology proved itself as suitable for our project since it allowed fast prototyping and easy adaptability to new requirements, while still capturing system specifications from the beginning of the project.

References

[1] Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer, 21*(5), 61–72.

[2] Cockburn, A. (1998). *Basic Use Case Template* (Technical Report No. 96). Humans and Technology.

[3] Dick, Jeremy., Hull, Elizabeth., Jackson, Ken. (2017). *Requirements engineering*. Springer.

[4] Grinberg, Miguel. (2018). *Flask web development: Developing web applications with Python*. O'Reilly Media, Inc.

[5] Kampmann, A., Alrifaee, B., Kohout, M., Wüstenberg, A., Woopen, T., Nolte, M., Eckstein, L., Kowalewski,S. (2019, October). A dynamic service-oriented software architecture for highly automated vehicles. In *2019*

[3] https://docs.gitlab.com/ee/ci/

IEEE Intelligent Transportation Systems Conference (ITSC) (pp. 2101–2108). IEEE. https://doi.org/10.1109/ITSC.2019.8916841

[6] Lampe, Bastian., Woopen, Timo., Eckstein, Lutz. (2019). Collective driving-cloud services for automated vehicles in UNICARagil. In *28. Aachen Colloquium Automobile and Engine Technology: October 7th–9th, 2019, Eurogress Aachen, Germany* (pp. 677–703). Institute for Automotive Engineering, RWTH Aachen University. https://doi.org/10.18154/RWTH-2019-10061

[7] Luckow, Andre., Cook, Matthew., Ashcraft, Nathan., Weill, Edwin., Djerekarov, Emil., Vorster, Bennie. (2016). Deep learning in the automotive industry: Applications and tools. In *2016 IEEE International Conference on Big Data (Big Data)* (pp. 3759–3768). IEEE.

[8] Olariu, Cristian., Assem, Haytham., Ortega, Juan Diego., Nieto, Marcos. (2019). A cloud-based AI framework for machine learning orchestration: A “driving or not-driving” case-study for self-driving cars. In *2019 IEEE Intelligent Vehicles Symposium (IV)* (pp. 1715–1722). IEEE.

[9] Paetsch, Frauke., Eberlein, Armin., Maurer, Frank. (2003). Requirements engineering and agile software development. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises* (pp. 308–313). IEEE.

[10] Rumpe, Bernhard. (2011). *Modellierung mit UML* (2nd ed.). Xpert.press. Springer Berlin.

[11] Rupp, Christine., et al. (2014). *Requirements-Engineering und -Management: Aus der Praxis von klassisch bis agil.* Carl Hanser Verlag GmbH Co KG.

[12] Schräder, Tobias., Stolte, Torben., Jatzkowski, Inga., Graubohm, Robert., Maurer, Markus. (2019). An approach for a requirement analysis for an autonomous family vehicle. In *2019 IEEE Intelligent Vehicles Symposium (IV)* (pp. 1597–1603). IEEE.

[13] Schwaber, Ken., Beedle, Mike. (2002). *Agile software development with Scrum* (Vol. 1). Prentice Hall Upper Saddle River.

[14] Woopen, Timo., Lampe, Bastian., Böddeker, Torben., Eckstein, Lutz., Kampmann, Alexandru., Alrifaee, Bassam., Kowalewski, Stefan., Moormann, Dieter., Stolte, Torben., Jatzkowski, Inga., Maurer, Markus., Möstl, Mischa., Ernst, Rolf., Ackermann, Stefan., Amersbach, Christian., Winner, Hermann., Püllen, Dominik., Katzenbeisser, Stefan., Leinen, Stefan., Becker, Matthias., Stiller, Christoph., Furmans, Kai., Bengler, Klaus., Diermeyer, Frank., Lienkamp, Markus., Keilhoff, Dan., Reuss, Hans-Christian., Buchholz, Michael., Dietmayer, Klaus., Lategahn, Henning., Siepenkötter, Norbert., Elbs, Martin., v. Hinüber, Edgar., Dupuis, Marius., Hecker, Christian.. (2018, October). UNICARagil – Disruptive modular architectures for agile, automated vehicle concepts (1st ed.). In *27. Aachener Kolloquium Fahrzeug- und Motorentechnik: October 8th–10th, 2018 – Eurogress Aachen* (pp. 663–694). Aachener Kolloquium Fahrzeug- und Motorentechnik GbR. https://doi.org/10.18154/RWTH-2018-229909