



The Connected Car in the Cloud

A Platform for Prototyping Telematics Services

Tobias Häberle, Lambros Charissis, Christoph Fehling, Jens Nahm, and Frank Leymann

As cars turn into computers on wheels, it becomes vital to experiment with novel application scenarios and to envision suitable abstractions for integrating cars into existing enterprise information systems. This insightful project report tells the story of a prototyping platform for connected-car software and shares the project's experience with a cloud-computing pattern language that helped drive the architectural platform design.
—*Cesare Pautasso and Olaf Zimmerman, department editors*

CONNECTED CARS are hitting the road. Almost every automotive company has a solution for a connected car.¹ We already use many connected devices in our daily life—for example, smartphones and tablets. The connected car adds to this portfolio with its own flavor.

This article shares our experiences delivering services for the connected car. We built the Connected-Car Prototyping Platform and reusable application templates to enable prototyping of telematics services. The platform and templates simplify prototype development and reduce time-to-market. They're based on cloud principles to achieve lower initial setup costs for prototypes and better scalability.

Addressing the Challenges of Connected Cars

Providing connected services—that is, apps—for smartphones has become simple owing to a range of frameworks and back-end platforms such as Parse (<https://parse.com>). Such frameworks and platforms offer software development kits for many mobile platforms and programming languages. They typically provide services such as data storage, hosting, and analytics to improve the development experience.

Connected-car platforms, however, present more challenges: longer life cycles (a car typically runs 10 or more years), less IT standardization, less reliable connectivity (typically, parked cars are offline), a bigger va-

riety of connectivity hardware, and so on. In addition, the automotive industry and the related multimedia industry are still unclear about customer demand. There's an ongoing discussion about the killer features for the next generation of connected cars.^{2,3} Without knowledge of future customer demand, time-to-market and flexibility become crucial.

To address these challenges and simplify development of telematics services, we built the Connected-Car Prototyping Platform. It provides a back end for applications interacting with cars. For us, the connected car is “just another” connected device similar to smartphones, watches, or even parking lots. The platform provides an abstraction of such

connected devices for developers. It also delivers services such as identity management and data storage—for example, for nonrelational data. These services are the foundation for many value-added services and applications. So, our platform offers a run-time environment for many telematics services. It also

telematics service prototypes, which are based on the templates.)

Our design process had five steps:

- *Decomposition* divides the application functionality into separate components.
- *Workload characterization* estimates component use over time.

To enable properties such as scalability, our design process considers cloud-computing patterns in five steps.

supports properties associated with cloud computing, such as self-service for developers and dynamic scalability of hosted applications. To enable such properties, we employed cloud-computing patterns⁴ to design the platform and develop a reference architecture for hosted applications. Developers can focus on application functionality and don't have to deal with infrastructure or middleware.

In addition, application templates provide architecture blueprints and code snippets, on which developers can base their application prototype implementations. The templates homogenize prototype design, further accelerating application development and simplifying runtime management.

To design the platform and templates, we applied Christoph Fehling and his colleagues' design process.⁵ Each step of this process considers a certain set of the cloud-computing patterns to use in the application architecture. (However, developers don't need this process to create

- *Data (state) management design* identifies stateful and stateless components.
- *Component refinement* designs, in detail, functionality for user interfaces, processing, and data access.
- *Elasticity and resiliency configuration* describes run-time behavior to address workload changes and faulty components.

The patterns we used are at www.cloudcomputingpatterns.org.

The Prototyping Platform

To design our platform, we first collected high-level functional and nonfunctional requirements as a baseline. We decomposed the platform functionality according to the supported domains (see *Domain-Driven Design: Tackling Complexity in the Heart of Software*⁶ for more information on domain-driven decomposition). This led to six application components (platform services): identity and access, the telematics service repository,

user profiles, billing, communication, and the virtual vehicle.

Figure 1 depicts these platform services and their data-handling capabilities. The services are identified to address initial functional needs. They may then be consumed by other value-added (telematics) services, applications, or user front ends—for example, to recommend the most suitable fuel station or parking lot on the basis of the vehicle's location, fuel status, and so on.

The architecture specifically addresses the high degree of uncertainty and the need for flexibility. In particular, workloads are unpredictable. They depend on factors such as the number of vehicles and users as well as possible resource-intensive applications running on the platform. So, we designed each platform service to handle unpredictable workloads.

The platform decomposes a service into a business logic tier and a data tier, separating the services' business logic from data storage. The separation of tiers and concerns allows independent scalability of each tier.

The business logic tier is the heart of each service, ensuring correct functionality. To ease scalability, this tier is stateless. State is either handled in the storage services or provided with each request to the tier. A central load balancer handles scaling.

Data, such as vehicle positions and service intervals, is stored in the data tier, which therefore is stateful. The data can be stored independently of applications or use cases. The consistency level at this tier varies, depending on the underlying service and its data classification.

For example, we designed the identity-and-access service using the Relational Database and Strict Consistency patterns because it holds

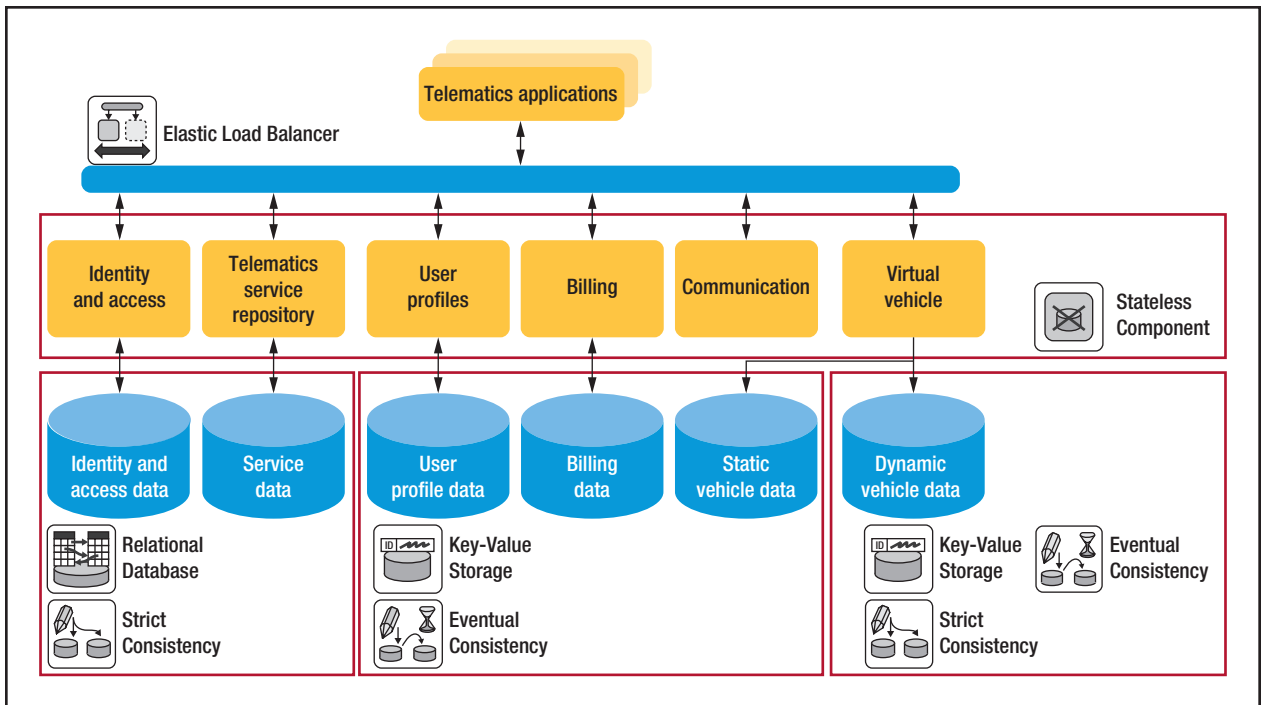


FIGURE 1. The Connected-Car Prototyping Platform architecture.⁷ Telematics applications are built on top of platform services, which are stateless. The applications' data tier handles state for the platform services. Data storage depends on data characteristics (for example, key-value storage for dynamic vehicle data).

complex data relationships. The definition of access policies is based on the association of users, vehicles, and services. Changes must be reflected immediately—for example, when service access is revoked for a user. Another example is the virtual-vehicle service, which provides an abstraction of vehicles. It stores dynamic vehicle data in a key-value format. For performance and availability reasons, the platform distinguishes data with strict consistency requirements (for example, door lock status) from data with eventual-consistency characteristics (for example, a GPS position as part of a GPS data stream). In our case, a platform service provides the data tier capabilities. So, the platform can automatically scale this tier.

At the project's start, we decided to base the implementation on open-source components. We did this to

- learn more about open source's advantages and drawbacks in an enterprise environment and
- initially decrease setup costs to verify the ideas.

We implemented the platform services in Java. Each service exposes a RESTful API over HTTP⁸ for interaction (REST stands for Representational State Transfer). We selected Java-Script Object Notation as the data format to support mobile applications. The business logic is deployed on a JBoss application server. We realized the data tier using HSQLDB (Hyper SQL Database) for relational data and Cassandra for nonrelational data. The platform applies Cassandra's tunable-consistency mode⁹ to satisfy different consistency requirements for nonrelational data.

The Application Templates

Our platform provides fundamental functionality that each telematics application prototype can use, thereby reducing complexity for developers. However, developers will still spend significant time

- setting up and integrating the prototype's fundamental architectural components and
- implementing code that's independent of specific application functionality and could be reused the same way in every application prototype.

So, to further speed up prototyping, we introduced the application templates.

The templates provide an architecture for telematics application prototypes that are from the same

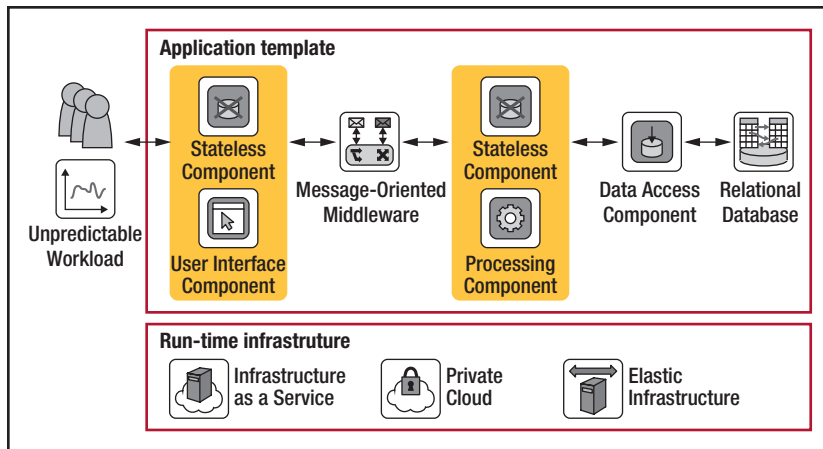


FIGURE 2. The architecture of an example application template. To provide maximum flexibility, the architecture is a modification of the Three-Tier Cloud Application pattern.

application domain and face similar requirements. An architecture by itself, however, will still require the developer to implement it. So, templates also offer the architecture's implementation. This implementation provides the required infrastructure and middleware components and is available for provisioning in one or more virtual server images.

Developers build application prototypes by provisioning templates in a prototyping environment and enriching them with individual functional code. So, they don't have to worry about finding a proper architecture and selecting, integrating, and provisioning feasible middleware products. Instead, they can focus on implementing the application functionality and generate added value faster.

The challenge when designing templates is to find an architecture that serves as a template for a possibly high number and variety of potential application prototypes. When trying to determine common requirements of application prototypes, we found we had to distinguish be-

tween two scenarios. The first scenario deals with data analysis. The vehicle frequently sends data to the connected-car platform; the data is stored persistently and then analyzed by a telematics application.

However, we focus here on the second scenario, in which telematics applications enable vehicle-user interaction. (At Daimler TSS, we frequently deal with this scenario.) With these applications, users can control a vehicle remotely, send information (such as locations) to the vehicle, or remotely view dynamic vehicle data such as the fuel state. Using this basic functionality, we can create advanced use cases that provide added value to users. For instance, by leveraging third-party APIs, users can select the most suitable gas station on the basis of the vehicle's fuel type, cruising range, current location, and destination, and the traffic volume.

To design a template for the second scenario, we first decomposed the template into three functional blocks: user interface, processing, and data access. We did this because applications in this scenario

typically require user interaction and the persistent storing of application-specific data, and must handle computationally intensive tasks. When you're designing application prototypes, it's often hard to foresee how intensively they'll be used and to which user groups they'll be exposed. So, the template architecture must be flexible enough to deal with unpredictable workloads. A further design consideration is the application data state. Here, we designed the user interface component and processing component to be stateless.

The template design resulted in implementation of a modified Three-Tier Cloud Application pattern. The three tiers might be exposed to different workloads. Depending on the application prototype use case, the processing component might have to deal with computationally intensive tasks that require scaling, or it might be used less frequently than the user interface component. To achieve independent scalability of the application components, we integrated them in a loosely coupled manner, using message-oriented middleware.

We discussed integrating the processing component and data access component in a loosely coupled way as well, as proposed by the Three-Tier Cloud Application pattern. However we didn't expect the data access component to face a high workload because the applications in this scenario rely mainly on platform services for data storage. So, we used remote procedure calls to access the data access component.

We employ a private cloud offered by a VMware product suite because it provides high availability of the virtual environment by replicating the underlying physical hardware. We didn't implement

high availability at the application level because we didn't consider it necessary for prototyping. So, elasticity and resiliency configuration occurs later, when a prototype is productized. The Three-Tier Cloud Application pattern forms the basis for such extensions.

Figure 2 illustrates the template's architecture. We implemented it using Java Platform Enterprise Edition technology (see Figure 3). The user interface component implements Java Servlets and is deployed on an Apache Tomcat webserver. The Apache ActiveMQ message-oriented middleware integrates the user interface and processing components (to achieve loose coupling). The template leverages the Java Message Service (JMS) API to provide an interface with the messaging provider. The processing tier and data-handling tier use the JBoss application server. The processing component can act as a Competing Consumer pattern¹⁰ by using Java message-driven beans. It also contains an HTTP client to utilize prototyping-platform services and access the data service. We implemented the data service as a RESTful webservice using JAX-RS (Java API for RESTful Web Services). The storage component is a MySQL database.

All templates are Maven-based Java projects stored in a central Maven repository. Ideally, more than one template is available for developers so that they can select the most suitable one for a specific application prototype scenario. For each template, corresponding virtual-machine images are available in the private cloud we mentioned earlier. These templates can be instantiated to provide the infrastructure and middleware required to deploy the template-based prototypes.

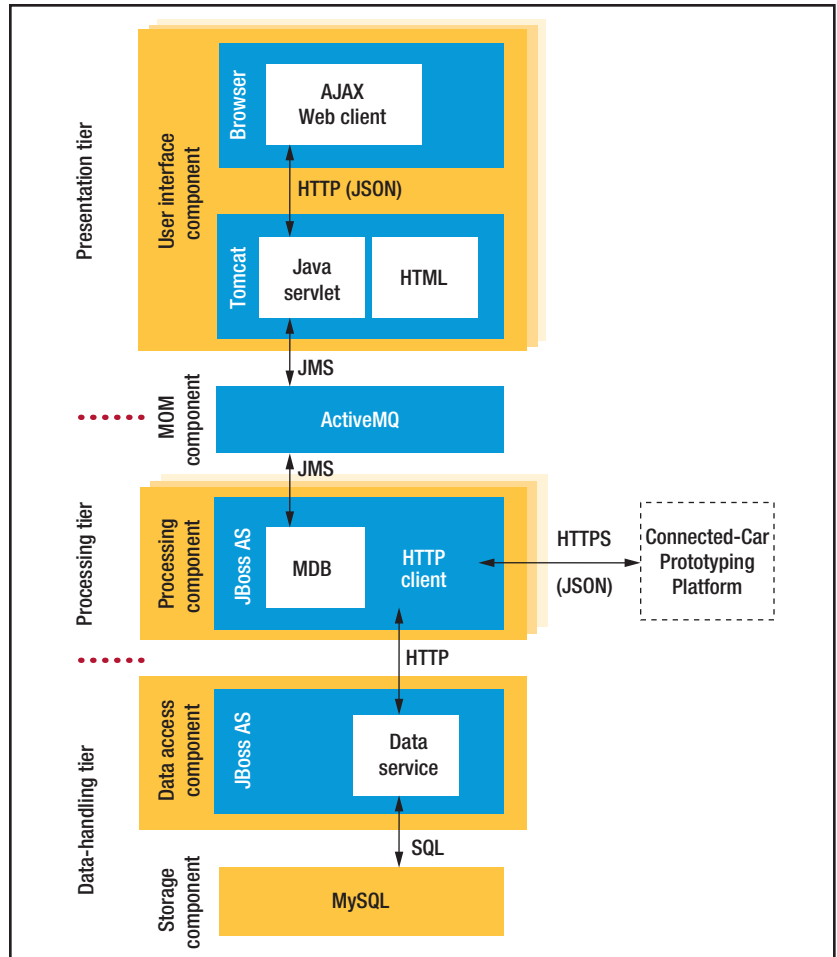


FIGURE 3. An implementation of the template in Figure 2. The implementation uses Java Platform Enterprise Edition technology. The user interface component and processing component are integrated in a loosely coupled manner using messaging, so that they can scale independently. The processing component contains the functionality to access the prototyping-platform services. AJAX is Asynchronous JavaScript and XML, JSON is JavaScript Object Notation, MOM is message-oriented middleware, JMS is Java Message Service, MDB is message-driven bean, and JBoss AS is JBoss Application Server.

We extended a framework prototype¹¹ to automate loading source code for templates into development environments. We further extended it to automate infrastructure and middleware provisioning and application deployment. For that, we implemented a jclouds (<https://jclouds.apache.org>) Maven plug-in to en-

able interaction with a private-cloud API. Maven plug-ins also perform middleware configuration and prototype deployment.

Discussion

We successfully used our platform to develop telematics services for a research-and-development group.

Overall, our pattern-based approach resulted in reusable functionality and guidelines that sped up development.

The current implementation is running successfully. Users can easily create services using templates. The

accessed in a linear form. Nevertheless, the patterns didn't address and solve all the architectural challenges we experienced. Patterns provide good hints on how to solve most challenges. However, they can still

Patterns provide hints on how to solve most challenges but can be incomplete or ambiguous.

architecture has been robust enough to accommodate new requirements without fundamental changes. For example, we added a near-real-time capability to stream data from a car to the Web for live monitoring. We extended the vehicle's connectivity unit and the platform with an offline capability that addresses cellular dead spots cars move through.

We found it's important to engage with IT stakeholders, especially IT operations staff, as early as possible to identify and avoid potential impediments. In the end, even small aspects might result in a complete showstopper owing to company policies or other corporate regulations. Some of our devices rely on non-HTTP protocols and nonstandard HTTP ports (ports other than 80 and 443) when communicating over the Internet—for example, using MQTT¹² as a transport protocol with custom ports. In an enterprise environment, the operations team will support such exceptions only if they're discussed at an early stage.

We also learned that patterns serve as a good guideline. Because they reference each other, the knowledge they provide doesn't have to be

be incomplete or ambiguous with many implementation options, or they might not cover all the requirements. The following four examples illustrate such situations.

First, in our case, the architecture serves only a little traffic at the beginning. However, it should be able to serve a rapidly growing number of vehicles and consumers—up to 5 million clients. The patterns we use enable the platform and deployed applications to scale as the number of users grows. However, additional challenges might arise—for example, if the amount of application data increases unexpectedly. So, in the future, the complete architecture might require adjustments.

The second example involves data consistency. To deal with a large amount of data (for example, a fleet of several million cars sending regular status events), the best practice is to partition the data into separate consistency levels—for example, strict versus eventual consistency. In our case, this approach requires different database technologies, which increases implementation and maintenance effort. The Data Access Component pattern re-

duces effort for application development and maintenance but not for database operations.


The use of different database technologies and consistency levels has led to new challenges affecting various stakeholders and supported business cases. The operations workforce must have a larger skill set to support different database technologies. Users evaluating the data must learn to acknowledge that it might not always be up to date. Finally, the business cases that such technologies support must take into account data inconsistencies. Patterns addressing these challenges seem a promising research area.

Third, the patterns cover how to use cloud offerings but not how to actually build a cloud. This is because they target application developers, not cloud providers. To design our platform, we modified the Three-Tier Cloud Application pattern, as we mentioned before. We added the central load balancer and designed the platform services as independent components. However, it would be helpful for cloud providers to have a pattern that provides initial guidelines.

The final example deals with extending the patterns to deal with wearable technology and the Internet of Things. Wearables such as smart watches are a rapidly growing market for connected services. These devices come with new interaction models and new technical characteristics. As with cars, such devices aren't always on; patterns dealing with queuing, caching, and synchronization would be beneficial. Also, one aspect of the Internet of Things is the growing number of sensors and actors. Patterns could help deal with the complexity and could address, for example, identity and con-

figuration management, or data aggregation and filtering.

Patterns don't solve all problems automatically. They can be misused if they're misunderstood. So, an experienced architect should validate the selected patterns for their applicability and perform a critical cross-check on the overall scenario.

We plan to create templates covering a variety of application scenarios and provide them to developers. Making the templates available in a central repository on the Internet will let developers select and download those that help them most in their current use case. The community will also be able to provide its own templates. Maven already proved to be a usable repository in the current setup. Its robustness and scalability in a broader setup—for example, for use by a larger group of developers—needs further investigation. 

Acknowledgments

Icons used in Figures 1 and 2:  <http://cloudcomputingpatterns.org>

References

1. “Gartner Says by 2020, a Quarter Billion Connected Vehicles Will Enable New In-Vehicle Services and Automated Driving Capabilities,” Gartner, 26 Jan. 2015; www.gartner.com/newsroom/id/2970017.
2. D. Hardware, “Buckle Up: The Connected-Car Revolution Is Almost Here,” *Engadget*, 13 Jan. 2015; www.engadget.com/2015/01/13/connected-car-revolution.
3. J. Carter, “How Your Car Will Use Online Apps,” *Techradar*, 24 Jan. 2013; www.techradar.com/news/car-tech/how-your-car-will-use-online-apps-1125966.
4. C. Fehling et al., *Cloud Computing Patterns*, Springer, 2014.
5. C. Fehling, F. Leymann, and R. Retter, “Your Coffee Shop Uses Cloud Computing,” *IEEE Internet Computing*, vol. 18, no. 5, 2014, pp. 52–59.
6. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
7. M. Hilbert, “Architecture for a Cloud-Based Vehicle Telematics Platform,” diploma thesis 3575, Univ. Stuttgart, 2013.
8. S. Allamaraju, *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*, O'Reilly, 2010.
9. “What Is Tunable Consistency?,” Planet Cassandra, 2015; http://planetcassandra.org/general-faq/#0.1_data-3.
10. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2003.
11. A. Schraitle, “Provisioning of Customizable Pattern-Based Software Artifacts into Cloud Environments,” diploma thesis 3468, Inst. of Architecture of Application Systems, Univ. Stuttgart, 2013.
12. *MQTT Version 3.1.1*, Oasis, 29 Oct. 2014; <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>.

TOBIAS HÄBERLE is a senior IT architect at Daimler TSS. Contact him at tobias.haerberle@daimler.com.

LAMBROS CHARISSIS is an IT architect at Daimler TSS. Contact him at lambros.charissis@daimler.com.

CHRISTOPH FEHLING is a research associate and PhD student at the University of Stuttgart's Institute of Architecture of Application Systems. Contact him at christoph.fehling@iaas.uni-stuttgart.de.

JENS NAHM is the manager of the Enterprise Architecture Team at Daimler TSS. Contact him at jens.nahm@daimler.com.

FRANK LEYMAN is a full professor of computer science at the University of Stuttgart and the director of the university's Institute of Architecture of Application Systems. Contact him at frank.leymann@iaas.uni-stuttgart.de.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.



Showcase Your Multimedia Content on Computing Now!

IEEE Computer Graphics and Applications seeks computer graphics-related multimedia content (videos, animations, simulations, podcasts, and so on) to feature on its homepage, www.computer.org/cga.

If you're interested, contact us at cga@computer.org. All content will be reviewed for relevance and quality.

IEEE Computer Graphics AND APPLICATIONS