


# Active-Code Replacement in the OODIDA Data Analytics Platform<sup>\*</sup>

Gregor Ulm<sup>1,2</sup><sup>[0000-0001-7848-4883]</sup>, Emil Gustavsson<sup>1,2</sup><sup>[0000-0002-1290-9989]</sup>,  
and Mats Jirstrand<sup>1,2</sup><sup>[0000-0002-6612-8037]</sup>

<sup>1</sup> Fraunhofer-Chalmers Research Centre for Industrial Mathematics,  
Chalmers Science Park, 412 88 Gothenburg, Sweden

<sup>2</sup> Fraunhofer Center for Machine Learning,  
Chalmers Science Park, 412 88 Gothenburg, Sweden  
{gregor.ulm, emil.gustavsson, mats.jirstrand}@fcc.chalmers.se  
<http://www.fcc.chalmers.se/>

**Abstract.** OODIDA (On-board/Off-board Distributed Data Analytics) is a platform for distributing and executing concurrent data analytics tasks. It targets fleets of reference vehicles in the automotive industry and has a particular focus on rapid prototyping. Its underlying message-passing infrastructure has been implemented in Erlang/OTP. External Python applications perform data analytics tasks. Most work is performed by clients (on-board). A central cloud server performs supplementary tasks (off-board). OODIDA can be automatically packaged and deployed, which necessitates restarting parts of the system, or all of it. This is potentially disruptive. To address this issue, we added the ability to execute user-defined Python modules on clients as well as the server. These modules can be replaced without restarting any part of the system and they can even be replaced between iterations of an ongoing assignment. This facilitates use cases such as iterative A/B testing of machine learning algorithms or modifying experimental algorithms on-the-fly.

**Keywords:** Distributed computing, code replacement, Erlang

## 1 Introduction

OODIDA is a modular system for concurrent distributed data analytics for the automotive domain, targeting fleets of reference vehicles [6]. Its main purpose is to process telemetry data at its source as opposed to transferring all data over the network and processing it on a central cloud server (cf. Fig. 1). A data analyst interacting with this system uses a Python library that assists in creating and validating assignment specifications. Updating this system with new computational methods necessitates terminating and redeploying software. However, we would like to perform updates without terminating ongoing tasks.

<sup>\*</sup> The final authenticated version is available online at [https://doi.org/10.1007/978-3-030-48340-1\\_55](https://doi.org/10.1007/978-3-030-48340-1_55).

We have therefore extended our system with the ability to execute user-defined code both on client devices (on-board) and the cloud server (off-board), without having to redeploy any part of it. As a consequence, OODIDA is now highly suited for rapid prototyping. The key aspect of our work is that active-code replacement of Python modules piggybacks on the existing Erlang/OTP infrastructure of OODIDA for sending assignments to clients, leading to a clean design. This paper is a condensed version of a work-in-progress paper [4], giving an overview of our problem (Sect. 2) and its solution (Sect. 3), followed by an evaluation (Sect. 4) and related work (Sect. 5).

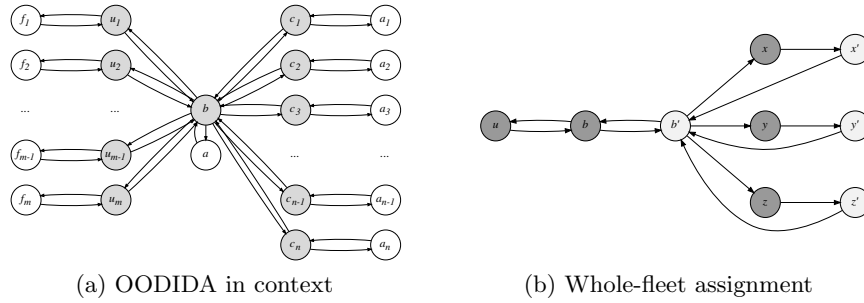


Fig. 1: OODIDA overview and details: In (a) user nodes  $u$  connect to a central cloud  $b$ , which connects to clients  $c$ . The shaded nodes are implemented in Erlang/OTP; the other nodes are external Python applications, i.e. the user front-ends  $f$ , the external server application  $a$ , and external client applications  $a$ . In (b) the core of OODIDA is shown with permanent nodes (dark) and temporary handlers (light) in an instance of a whole-fleet assignment. Cloud node  $b$  spawned an assignment handler  $b'$ . After receiving an incoming task, clients  $x, y$  and  $z$  spawned task handlers  $x', y'$ , and  $z'$  that interact with external applications. Nodes  $x$  and  $x'$  correspond to  $c_1$  in (a) etc.

## 2 Problem

OODIDA has been designed for rapid prototyping, which implies that it frequently needs to be extended with new computational methods, both for on-board and off-board data processing. To achieve this goal, Python applications on the cloud and clients have to be updated. Assuming that we update both, the following steps are required: The user front-end  $f$  needs to be modified to recognize the new off-board and on-board keywords for the added methods, including checks of assignment parameter values. In addition, the cloud and client applications have to be extended with the new methods. All ongoing assignments need to be terminated and the cloud and clients shut down. Afterwards, we can redeploy and restart the system. This is disruptive, even without taking into account potentially long-winded software development processes in large organizations. On the other hand, the turn-around time for adding custom methods would be

much shorter if we could do so at runtime. Active-code replacement targets this particular problem, with the goal of further improving the suitability of OODIDA for rapid prototyping.

### 3 Solution

With active-code replacement, the user can define a custom Python module for the cloud and for client devices. It is implemented as a special case of an assignment. The front-end  $f$  performs static and dynamic checks, attempting to verify correctness of syntax and data types. If these checks succeed, the provided code is turned into a JSON object and ingested by user node  $u$  for further processing. Within this JSON object, the user-defined code is stored as an encoded text string. It is forwarded to cloud node  $b$ , which spawns an assignment handler  $b'$  for this particular assignment. Custom code can be used on the cloud and/or clients. Assuming clients have been targeted with active-code replacement, node  $b'$  turns the assignment specification into tasks for all clients  $c$  specified in the assignment. Afterwards, task specifications are sent to the specified client devices. There, the client process spawns a task handler for the current task, which monitors task completion. The task handler sends the task specification in JSON to an external Python application, which turns the given code into a file, thus recreating the Python module the data analyst initially provided. The resulting files are tied to the ID of the user who provided it. After the task handler is done, it notifies the assignment handler  $b'$  and terminates. Similarly, once the assignment handler has received responses from all task handlers, it sends a status message to the cloud node and terminates. The cloud node sends a status message to inform the user that their custom code has been successfully deployed. Deploying custom code to the cloud is similar, the main difference being that  $b'$  communicates with the external Python application on the cloud.

If a custom on-board or off-board computation is triggered by a special keyword in an assignment specification, Python loads the user-provided module. The user-specified module is located at a predefined path, which is known to the Python application. The custom function is applied to the available data after the user-specified number of values has been collected. When an assignment uses custom code, external applications reload the custom module with each iteration of an assignment. This leads to greater flexibility: Consider an assignment that runs for an indefinite number of iterations. As external applications can process tasks concurrently, and code replacement is just another task, the data analyst can react to intermediate results of an ongoing assignment by deploying custom code with modified algorithmic parameters while this assignment is ongoing. As custom code is tied to a user ID, there is furthermore no interference due to custom code that was deployed by other users. The description of active-code replacement so far indicates that the user can execute custom code on the cloud server and clients, as long as the correct inputs and outputs are consumed and produced. What may not be immediately obvious, however, is that we can now

create *ad hoc* implementations of even the most complex OODIDA use cases in custom code, such as federated learning [3].

Inconsistent updates are a problem in practice, i.e. results sent from clients may have been produced with different custom code modules in the same iteration of an assignment. This happens if not all clients receive the updated custom code before the end of the current iteration. To solve this problem, each provided module with custom code is tagged with its md5 hash signature, which is reported together with the results from the clients. The cloud only uses the results tagged with the signature that achieves a majority. Consequently, results are never tainted by using different versions of custom code in the same iteration.

## 4 Evaluation

The main benefit of active-code replacement is that code for new computational methods can be deployed right away and executed almost instantly, without affecting other ongoing tasks. In contrast, a standard update of the cloud or client installation necessitates redeploying and restarting the respective components of the system. In an idealized test setup, where the various workstations that run the user, cloud and client components of OODIDA are connected via Ethernet, it takes a fraction of a second for a custom on-board or off-board method to be available for the user to call when deployed with active-code replacement, as shown in Table 1. On the other hand, automated redeployment of the cloud and client installation takes roughly 20 and 40 seconds, respectively. The runtime difference between a standard update and active-code replacement amounts to three orders of magnitude. Of course, real-world deployment via a wireless or 4G connection would be slower as well as error-prone. Yet, the idealized evaluation environment reveals the relative performance difference of both approaches, eliminating potentially unreliable data transmission as a source of error.

This comparison neglects that, compared to a standard update, active-code replacement is less bureaucratic and less intrusive as it does not require interrupting any currently ongoing assignments. Also, in a realistic industry scenario, an update could take days or even weeks due to software development and organizational processes. However, it is not the case that active-code replacement fully sidesteps the need to update the library of computational methods on the cloud or on clients as OODIDA enforces restrictions on custom code. For instance, some parts of the Python standard library are off-limits. Also, the user cannot install external libraries. Yet, for typical algorithmic explorations, which users of our system regularly conduct, active-code replacement is a vital feature that increases user productivity far more than the previous comparison may imply. That being said, due to the limitations of active-code replacement, it is complementary to the standard update procedure rather than a competitive approach.

Table 1: Runtime comparison of active-code replacement of a moderately long Python module versus regular redeployment in an idealized setting. The former has a significant advantage. Yet, this does not factor in that a standard update is more invasive but can also be more comprehensive. The provided figures are the averages of five runs.

	Cloud	Client
Active-code replacement	20.3 ms	45.4 ms
Standard redeployment	23.6 s	40.8 s

## 5 Related Work

The feature described in this paper is an extension of the OODIDA platform [6], which originated from `f1-erl`, a framework for federated learning in Erlang/OTP [5]. In terms of descriptions of systems that perform active-code replacement, Polus by Chen et al. [1] deserves mention. A significant difference is that it replaces larger units of code instead of isolated modules. It also operates in a multi-threading environment instead of the highly concurrent message-passing environment of OODIDA. We also noticed a similarity between our approach and Javelus by Gu et al. [2]. Even though they focus on updating a stand-alone Java application as opposed to a distributed system, their described "lazy update mechanism" likewise only has an effect if a module is indeed used. This mirrors our approach of only loading a custom module when it is needed.

**Acknowledgements.** This research was financially supported by the project On-board/Off-board Distributed Data Analytics (OODIDA) in the funding program FFI: Strategic Vehicle Research and Innovation (DNR 2016-04260), which is administered by VINNOVA, the Swedish Government Agency for Innovation Systems. It took place in the Fraunhofer Cluster of Excellence "Cognitive Internet Technologies." Simon Smith and Adrian Nilsson helped with a supplementary part of the implementation of this feature and carried out the performance evaluation. Ramin Yahyapour (University of Göttingen) provided insightful comments during a poster presentation.

## References

1. Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.C.: Polus: A powerful live updating system. In: 29th International Conference on Software Engineering (ICSE'07). pp. 271–281. IEEE (2007)
2. Gu, T., Cao, C., Xu, C., Ma, X., Zhang, L., Lu, J.: Javelus: A low disruptive approach to dynamic software updates. In: 2012 19th Asia-Pacific Software Engineering Conference. vol. 1, pp. 527–536. IEEE (2012)
3. McMahan, H.B., Moore, E., Ramage, D., Hampson, S., et al.: Communication-efficient learning of deep networks from decentralized data. arXiv preprint [arXiv:1602.05629](https://arxiv.org/abs/1602.05629) (2016)

4. Ulm, G., Gustavsson, E., Jirstrand, M.: Facilitating Rapid Prototyping in the OODIDA Data Analytics Platform via Active-Code Replacement. arXiv preprint [arXiv:1903.09477](https://arxiv.org/abs/1903.09477) (2019)
5. Ulm, G., Gustavsson, E., Jirstrand, M.: Functional federated learning in Erlang (ffl-erl). In: Silva, J. (ed.) Functional and Constraint Logic Programming. pp. 162–178. Springer International Publishing, Cham (2019)
6. Ulm, G., Gustavsson, E., Jirstrand, M.: OODIDA: On-board/off-board distributed data analytics for connected vehicles. arXiv preprint [arXiv:1902.00319](https://arxiv.org/abs/1902.00319) (2019)