

# Cloud Process Execution Engine: Architecture and Interfaces

Juergen Mangler<sup></sup> and Stefanie Rinderle-Ma<sup></sup>

Department of Informatics, Technical University of Munich,  
85748 Garching, Germany  
{firstname.lastname}@tum.de

**Abstract** Process Execution Engines are a vital part of Business Process Management (BPM) and Manufacturing Orchestration Management (MOM), as they allow the business or manufacturing logic (expressed in a graphical notation such as BPMN) to be executed. This execution drives and supervises all interactions between humans, machines, software, and the environment. If done right, this will lead to a highly flexible, low-code, and easy to maintain solution, that allows for ad-hoc changes and functional evolution, as well as delivering a wealth of data for data-science applications.

The Cloud Process Execution Engine CPEE.org implements a radically distributed scale-out architecture, together with a minimal set of interfaces, to allow for the simplest possible integration with existing services, machines, and existing data-analysis tools.

Its open-source components can serve as a blueprint for future development of commercial solutions, and serves as a proven testbed for academic research, teaching, and industrial application since 2008.

In this paper we present the architecture, interfaces that make CPEE.org possible, as well as discuss different lifecycle models utilized during execution to provide overarching support for a wide range of data-analysis tasks.

## 1 Introduction

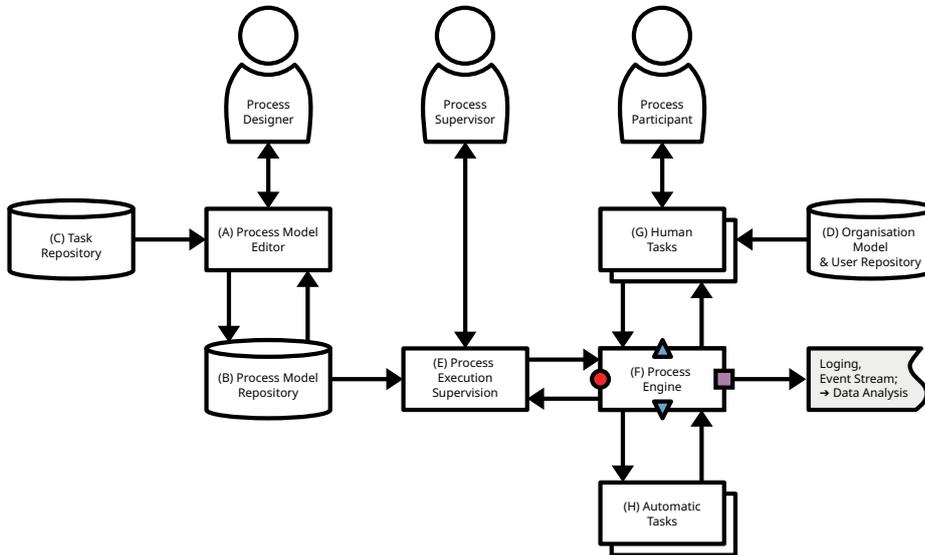
The Cloud Process Execution Engine is an open-source bare-bones radically service-oriented process engine, that, together with a set of components forms a Business Process Management (BPM) system that proved (a) great for teaching as all internal mechanisms are exposed as REST interfaces [2, 3] and can be inspected, used and augmented by interested students, and proved (b) great for research to either experiment with the currently prevalent graphical modelling language - Business Process Management notation (BPMN) - through extensions, or by developing altogether novel modelling languages [5]. Implementing worklists, correlators, run-time data-analysis, self-healing processes, or novel means of inter-process and inter-instance synchronization is easy and streamlined: external REST-services utilizing your language/framework of choice do it

all. Finally, (c) also companies took an interest, due to scalable highly flexible architecture that scales from a raspberry-pi with some instances to mainframes with 1000s of parallel running processes, while efficiently utilizing multi-core architectures.

CPEE.org tries to further the low-code and model-based process execution paradigm, that allows non-programmers to connect software, machines, and humans in simple and easy to understand ways. By allowing for ad-hoc instance changes to realize repair, as well as providing tools for process model versioning and evolution, it wants to show-case features that will hopefully make it into many current and future BPMs.

## 2 BPM Basics

Since many years, moving infrastructure and with it software components to the cloud is an important topic when dealing with digitization. Business Process Management is about graphical models containing sequences of activities, decisions and parallel branches. Activities describe how to invoke (external) functionality implementing an activity, including the required input, and how to transform the expected output to be usable for subsequent activities and decisions. Business Process Management (BPM) traditionally has been relying on monolithic Process Engines, mostly written in Java, conceived in the late 90s, and not changed much since then. The typically consist of the following tightly coupled components as also depicted in Fig. 1:



**Fig. 1.** Architecture & Stakeholders

- (A) Process Model Editor (UI, nowadays probably BPMN 2.x, CMMN, DMN)
- (B) Process Model Repository
- (C) Task Repository
- (D) Organization Model and User Repository
- (E) Process Execution Supervision (UI)
- (F) Process Engine
- (G) Worklists, Dashboards: Human Tasks (UIs)
- (H) Invoked Applications: Automatic Tasks

Thus the traditional stakeholders in a BPM system are the

- **Process Designers:** they create the process at design time, and improve / evolve process models if necessary.
- **Process Supervisors:** instantiate processes and supervise their execution. They user
- **Process Participants:** take care of the work as modelled by human / manual tasks. They user their own independent UI, and potentially know nothing about the existence of a BPM.

The Process Model Editor (**A**) allows Process Designer to create and change Process Models (PMs), which are stored in a Process Model Repository (**B**).

A PM is not executable, until each activity is assigned the corresponding functionality, and the required input/output parameters are set. The same goes for events. For example a timer event requires additional information how long to wait in machine-readable form. Typically the Task Repository (**C**) holds a list of functionalities available to the Process Designer for association with activities. This is true for automatic (**H**) as well as human tasks (**G**).

Human tasks (**G**) are typically come in the form of UIs called worklists or dashboards (see Sec. 5, “Components”), worklists require information about their users for work assignment. When worklists target work-distribution in organizations, user/role relationships are typically utilized to automatically distribute work between all users of a role. If worklists target customers, all work is assigned to one customer. In both cases users have to be logged-in/identified. The same goes for dash-boards if they allow for interaction, but it might also possible that interactions are possible without being logged in, because of the assumption that only eligible users have physical access to the dashboard. All information about users and roles (Organizational Structure) is kept in a Organization Model and User Repository (**D**).

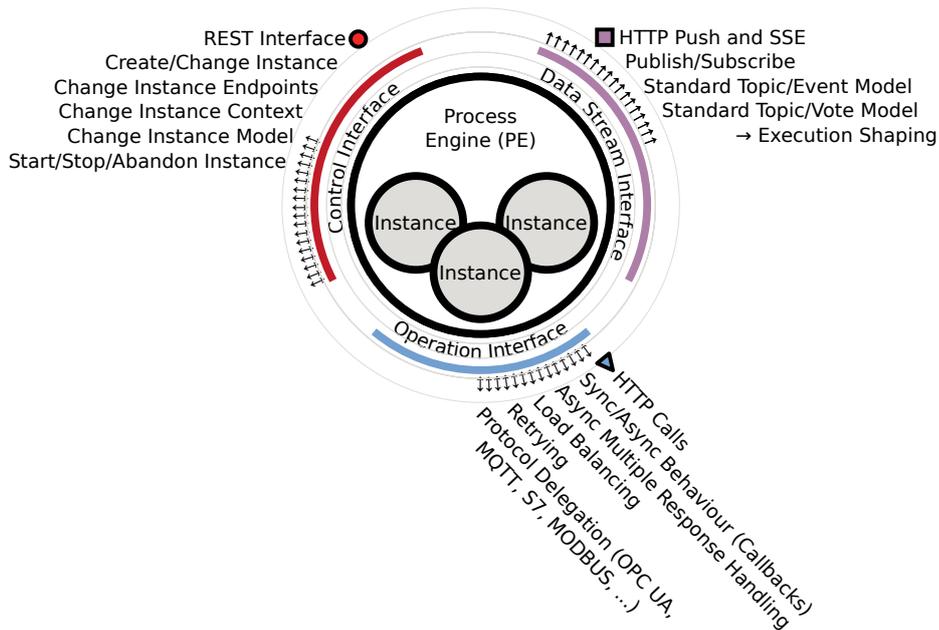
The Process Execution Supervision (UI) (**E**) is used to deploy PMs from the Process Model Repository to the Process Engine (**F**) - the is used to **create an instance (instantiate)**. Each instance is a **COPY** of the model, changes

to the instance might be possible (depending on engine features, i.e. run-time adaptation), in which case the model of the instance might deviate from its parent.

The Process Engine (PE) (**F**), is in charge of executing the instance model, and realizing all the invocations of external services or functionalities which are represented by activities, events or gateways (furthermore referred to as activity enactment). Process Engines most commonly are interpreters - just like for example the Java Virtual Machine (HotSpot). Other engines like CPEE.org are transpiling the model to lower-level languages for compilation/execution.

Invoked applications (**H**) are either typically either realized as Java components, that can be loaded into the process engine, or as external services, that have to implement a certain API as provided by the BPM software provider (e.g. Camunda provides APIs for Java, and JavaScript, unofficial Python support exists<sup>1</sup>). Depending on the PE these services might either be realized as REST/SOAP/OPC-UA/...

### 3 Process Engine Interfaces



**Fig. 2.** Interfaces of CPEE.org

<sup>1</sup> Last checked: 2022-09-25

The **PROCESS ENGINE (PE)** is at the heart of the architecture. All other components contribute data that is required for the execution of an instance, but the process engine executes the process and coordinates the interaction of all components, no matter if they are standardized ((**E**), (**G**)) or user-created ((**G**), (**H**)).

While the architecture given in Fig. 1, might be either monolithic or partially monolithic (components may be part of one big software package, either desktop-based or web-based), it is also possible to separate all components through clearly defined interfaces. The architecture of CPEE.org takes the second approach, and is realized as a set of loosely coupled services. Note that this is not necessary for the (**A**) Process Model editor, or any other editor for that matter, as e.g. a Task Editor only contributes to the (**C**) Task Repository, and an Organization Editor only contributes to (**D**) Organization Model & User Repository. So all editors interface if the engine through the data structures they produce ex-ante. These data-structures might be BPMN 2.0 Interchange format or CPEE.org trees for models, custom LDAP structures for Organization Models & User Lists, and proprietary lists for tasks.

So at runtime, the process engine really has three interfaces with active components, as marked through ● ▲ ■ as depicted in Fig. 1 and Fig. 2.

### 3.1 Control Interface

● The Control Interface allows UIs used by the Process Supervisor to perform all operations related to creating, starting and managing process instances.

In order to **create an instance**, a Process Execution Supervision (PES) UI has to allow to, (a) select a process model, (b) instantiate it, (c) and supervise its execution through an execution engine:

- Which activities are currently enacted/running?
- What is the current process context (the data-elements that exist during execution)?
- Which sub-processes have been spawned by the instance?

CPEE.org is an adaptive process engine, thus when an instance fails (stops), through the PES the following operations can be performed:

- Change endpoints for activities, i.e. change the functionality an activity invokes during enactment. E.g., when one production machine fails a second one might take over a task.
- Changing the process context, i.e. activities might yield faulty data that would prevent the successful execution of their instance. Manual changes to data-elements might save the instance.

- Change the thread of control, i.e. change which activities are enacted next. This might include skipping activities, but also re-doing activities. While re-enacting activities, or skipping activities might be harmful, as tasks typically have consequences, it might also be possible that re-enacting/skipping might save the instance. Thus any process engine should allow a Process Supervisor to do both.
- Re-starting the instance execution if it stops. As functionalities implementing activities can be temporary down, or endpoints/dataelements/thread of control can be changed to alleviate problems, re-starting the execution at a certain point is beneficial.
- Change the instance model, i.e. whenever a process instance is stopped, it might be necessary to change the instance model (e.g. repair the instance by inserting or deleting activities). Changed instance model become singletons - the no longer are identical to the process model they have been initially instantiated from.

CPEE.org strictly relies on a REST-interface to achieve all these changes.

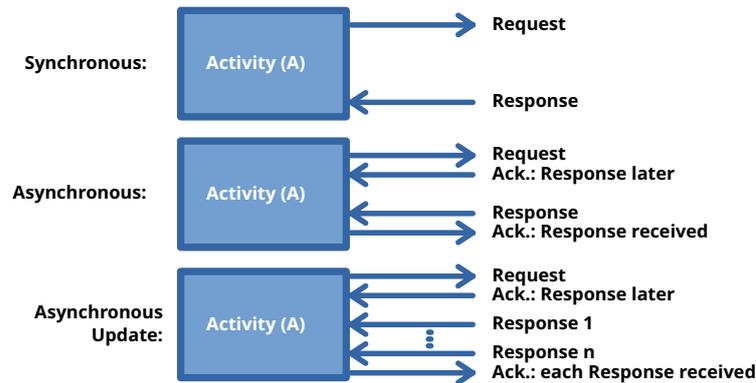
### 3.2 Operation Interface

▲ The purpose of the operation interface is to delegate and monitor the work described by activities. Activities (A) are modelling elements in BPMN (or other graphical notations), that describe how to invoke (external) functionality (F), including the required input, and how to transform the expected output to be usable for subsequent activities and decisions. By default the functionality is assumed to be a black box, from the point of view of a process engine (PE) it is not important what is going on inside. In other words: the PE manages the data-flow to and from these activities.



Many process engines rely on an API to implement the functionality invoked activities, i.e., F is implemented using an API, which allows to either (1) load F into the PE (old-school monolithic engines), or (2) start F as a server which can than be invoked by the PE enacting an activity. This has the advantage that the protocol utilized between A and F is not important and can be changed without affecting the implementation of F.

Alternatively, CPEE.org and other engines rely on protocol extensions instead of an API to implement functionalities. CPEE.org's primary protocol implementation is HTTP, which is extended by a set of CPEE.org-specific HTTP headers allow for some special interaction patterns between PE and F:



For the **synchronous pattern**, answers are returned immediately, which in HTTP is only possible if the answer is returned in a certain time-frame (about 30 seconds for normal network infrastructure). If the time-frame can not be satisfied by F, the PE upon enactment of the activity in a certain instance will receive a timeout, and the instance will be thus stopped. This type of interaction can therefore only be used for simple and fast interactions.

The **asynchronous pattern** describes that (F) can delay the answer for as long as necessary. This is done by telling the PE that the answer will arrive later. This only works if a callback address is available to F. The PE maintains the list of callback addresses. Each callback address allows the PE to forward the answer to a certain activity in a certain instance.

Finally, the **asynchronous update pattern**, describes a special case of the asynchronous pattern which allows to call back multiple times, e.g., to return a series of status updates to the activity, or an arbitrary number of data chunks (which is especially useful if large amounts of response data have to be handled by T). This works by adding a flag to each answer, telling the PE if further answers are to be expected.

All this is enabled by the addition of a minimal set of CPEE.org specific HTTP headers, as enumerated below. Common CPEE.org HTTP headers, sent with each request are:

- **CPEE-BASE** - base location of the engine where the instance is running on  
(e.g., <https://cpee.org/flow/engine/>)
- **CPEE-INSTANCE** - instance number  
(e.g., 123)
- **CPEE-INSTANCE-URL** - url pointing to the instance  
(e.g., <https://cpee.org/flow/engine/123>)

- **CPEE-INSTANCE-UUID** - unique identifier of the instance (e.g., 059a4f32-dcb1-4ad0-a700-ddd3d1fbf64f)
- **CPEE-CALLBACK** - url to send any information to, should the implementation decide to answer asynchronously (e.g., <https://cpee.org/flow/engine/123/callbacks/f8c24f12-1419>)
- **CPEE-CALLBACK-ID** - unique identifier for the answer (e.g., f8c24f12-1419)
- **CPEE-ACTIVITY** - id of the activity invoking a functionality (e.g. a1)
- **CPEE-LABEL** - label of the activity invoking a functionality (e.g. Query Production Schedule)

Each response (independent of the pattern) can sent the following optional headers:

- **CPEE-SALVAGE** - F communicates that it can currently not provide any answer, but might be available again later. This can be utilized by the PE in a fail-over scenario, to reroute the request to a different F or to retry the original F at a later point in time. If this header is present, its value is expected to be always “true”.
- **CPEE-INSTANTIATION** - F communicates that it has instantiated a (sub-) process. F will most probably additionally return the instance-url in the body of the response (e.g., <https://host2.cpee.org/flow/engine/124/>). If this header is present, its value is expected to be always “true”.
- **CPEE-EVENT** - F communicates that it a functionality-custom event should be included in the data sent out trough the  data stream interface. This is especially useful if F has an internal lifecycle (e.g., if F implements a worklist) and wants to signal custom lifecycle transitions (such as a user taking or giving back a task). If this header is present, its value is expected to carry the name of the custom signal (e.g., worklist/task-taken).

The **asynchronous pattern**, in addition to the three optional common response headers, has to use the **CPEE-CALLBACK** header with the “Ack.: Response later” message. If this header is present, its value is expected to be always “true”, and the PE will not continue the execution of the instance, but instead wait for a reply. Each HTTP PUT to the **CALLBACK-URL** will prompt the PE to forward the response to the activity and subsequently continue the instance.

The **asynchronous update pattern**, in addition to the three optional common response headers, has to use the **CPEE-CALLBACK**, exactly the same as the **asynchronous pattern**. For each response, additionally the **CPEE-UPDATE** header is to used. Whenever this header is present and its value is true, the PE forwards the message to a certain activity in a certain instance, but the instance is not allowed to continue, and the activity continues waiting for further responses. A response missing the **CPEE-UPDATE** header is con-

sidered the last response, thus the PE will forward the response to the activity and subsequently continue the instance.

Through these simple protocol extension, CPEE.org can support arbitrary interactions. Custom protocols, such as OPC-UA (i.e., machine interfaces), can be implemented as proxy F's. While the communication between A and F utilizes the mechanisms described above, F will communicate with third-party services and machines through custom protocol implementations.

### 3.3 Data Stream Interface

■ The purpose of the data stream interface is for a process engine PE to communicate the state of **instances (I)** as well as the state of **activities (T)** to micro-services connected to the interface.

The interface supports two ways a state communication: HTTP push to dedicated URLs, and HTTP server sent events (SSE) upon request. In order to communicate the state to external services, a PUB/SUB mechanism exists, that allows to subscribe to a certain sub-set of events. Each subscription has to carry:

- An optional endpoint, that denotes where to push the events. If the endpoint is omitted, it is assumed that the messages grouped by the subscription will be sent through SSE<sup>2</sup> upon explicit request.
- An set of topics and events, that describe certain aspects of the execution of an instance.

Execution aspects, furthermore called topics, are:

- The **state** topic contains a set of events describing potential instance states (see Sec. 4.1).
- The **activity** topic contains a set of events describing the state of an activity in a particular instance (see Sec. 4.2). Each activity is represented by a series of events, as an activity at least starts and finishes. As an instance might contain activities that are enacted in parallel, each event has to carry the activity id (e.g., t1). When an activity is enacted in a loop, the activity id is not enough to identify events belonging together, as through the network-based nature of event dispersal events might be out-of-order. Thus each enactment of an activity has to carry (in addition to the activity id) a unique enactment identifier, e.g., “t1-enactment-1” or “t1-enactment-2”.
- The **position** topic allows to monitor the progression between activities. This includes events when activities become active, activities are no longer

<sup>2</sup> SSE is actual a special kind of pull, where a client initiates a connection with a server, the server keeps the connection alive by heart-beating to the client, and can thus push data through the connection when it is available. It is thus a form of long-polling.

active, as well as events detailing the transition between two activities. A transition between two activities does not mean that they are in sequence, a transition might occur between an activity and a next activity based on a decision, or multiple activities might become active due to a parallel split.

- The **status** topic allows to monitor information about semantic execution properties of an instance, e.g., if a instance currently runs normally, or if some exception handling logic is active. The instance status can be changed as the result of any activity enactment.
- The **dataelements** topic allows to monitor the data-flow, independently of the control-flow of an instance. While the enactment of activities might change the process context (dataelements, variables), not each activity does so. Each event includes information about added, deleted and changed (from value, to value) dataelements.
- The **description** topic allows to monitor changes to the instance model. When ever an instance **is not running** (e.g., before an instance is started or when a instance stopped due to an error), changes/repairs to the process model might be applied. Changes can include assigning different functionality to an activity, inserting or deleting activities.
- The **endpoints** topic allows to monitor when a process instance links to new functionality. CPEE.org for each instance manages a list endpoints key/value pairs where each functionality is referenced by a key, e.g. timeout →<https://cpee.org/functionalities/timeout/>. Functionality is assigned to activities by this key. Change events can occur both at runtime (while an instance is executed) as well as while an instance is stopped. An activity might as part of their enactment dynamically change/adapt the endpoint list, namely changing the value of any key, resulting in activities invoking different functionality. This can be utilized to, for example, implement load-balancing or load-distribution. Each event includes information about added, deleted and changed (form, to) endpoints.
- The **attributes** topic allows to monitor changes to an instances attributes. Attributes might include the UUID of the process model an instance was originally spawned from (although the model of the instance might have changed), or an arbitrary number of labels and information assigned to the instance (name, author of the model, user responsible for repair, ...). Each event includes information about added, deleted and changed (form, to) attributes.
- The **condition** topic allows to monitor any decisions taken during the execution of an instance. This might include decisions taken based on xor/or gateways or loops. The event includes the condition, all involved dataelements and their values, as well as the result of the evaluation (true or false).
- The **task** topic groups a set of special and user-defined events, such as task/instantiation which is sent by functionalities implementing the creation

instantiation of sub-processes instances. Such events are sent by functionalities through the ▲ operation interface (see Sec.5.3) and are subsequently distributed to subscribed services through the task topic. This interface is particularly useful for communicating the lifecycle or application state of functionalities through a PE. For example, worklists/tasklists are just ordinary functionalities invoked as part of activity enactment. Although potentially a black box to the PE, worklists might have a fine-grained internal lifecycle dealing with how work is assigned to users, work on by users, including dealings with deadlines and conflicts. For runtime or ex-post data-analysis [9,10] it can be very useful to include this information in subscribable data streams dispersed by the PE.

By selecting from this broad menu of topics external services can analyse all aspects of an instance execution, regarding both data-flow and control-flow. Each subscriber can use the data to, for example, write fine-grained execution logs which include information far beyond the aspects specified in standards such as the XES<sup>3</sup> standard.

In addition to distributing events the interface also supports **execution shaping** [4], which allows external services subscribed to events to influence the execution, without invoking the ● control interface, but as part of the subscription to the ■ data stream interface.

While normal events are sent by the PE without waiting or acknowledging a response (fire and forget), special events, furthermore called votes are treated differently. The PE waits for a response from each subscribed external service and acts upon the responses. A minimal set of responses currently implemented by CPEE.org includes:

- **ack**: don't care or approval. Instance might continue to be executed as per the model.
- **callback**: answer will be sent later. Instance will remain in state running, but the activity referenced by the vote will remain frozen until the answer is received.
- **skip**: instance will remain in state running, the activity referenced by the vote will be skipped.
- **stop**: instance is stopped immediately.
- **start**: instance is started immediately.
- **value**: (1) the condition referenced in the vote is evaluating to the value (true, false). (2) the dataelement, endpoint, attribute referenced in the vote is set to the values.

Callback is the special case, that just delays the decision. All other responses have to be unambiguous, with ack being the neutral response. Examples

<sup>3</sup> <https://xes-standard.org/>

- If 1..n services send skip and the rest of the services send ack, skip goes into effect.
- The same rule is applicable to stop and start.
- If 1..n services respond with action, but disagree on true/false, the rest of the services send ack, then the instance will be stopped.
- If 1..n services respond with value, but disagree on the actual value, the rest of the services send ack, then the instance will be stopped.
- If 1..n services respond with the same value, the rest of the services send ack, then the value will be set and the instance resume executing.
- Responses of value, skip, start/stop can be partially combined:
  - value and skip can be combined, with value being enacted first, then the skipping the activity.
  - value and start/stop can be combined with value being enacted first, then starting/stopping the instance.
  - skip and stop can be combined, first the skipping the activity then stopping the instance.
  - start and skip can be combined, first starting the instance, the skipping the active activity immediately.
  - start, value, skip can be combined according to the schema above.
  - value, skip, stop can be combined according to the schema above.
- Dissenting start/stop responses can not be combined, the current state will remain.

Topics that have votes include:

- **state** topic: start / stop can be prohibited or allowed. This is useful when implementing model checking techniques. Furthermore external services can change endpoints, attributes and dataelements on start or stop through the value response.
- **dataelements, endpoints, attributes** topics: changes to individual can be blocked (action) or corrected (value) . Furthermore, an execution of an instance can be stopped (stop response) in compliance checking scenarios.
- **condition** topic: the evaluation of conditions can be modified with the value response. Again the instance can be stopped (stop response) if necessary.
- **description** topic: Individual changes to the model can be prohibited through action responses.

With this powerful voting mechanism runtime conformance and compliance checking, as well as self-healing, which all require not only certain data, but

also a set of actions to influence the execution, can be implemented through external services.

The alternative would be, to allow external services to utilize the ● control interface, which would entail to always stop instances before changes, in order to avoid race conditions. CPEE.org can thus cover the most important areas of runtime process mining (discovery can be ignored in this context) and adaptive process execution.

## 4 Lifecycles

In order to give a more detailed introduction to the **state**, **activity** and **task** topics introduced as part of the ■ data stream interface in the previous section, this section will discuss the lifecycle models for:

- 4.1: The execution of instances.
- 4.2: The enactment of activities.
- 4.3: The internal behaviour of worklist functionalities.

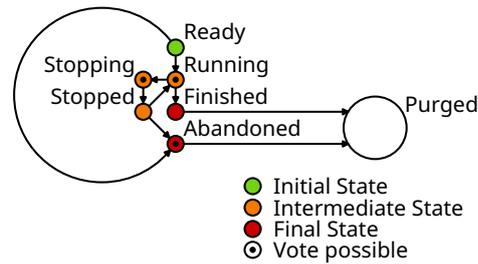
### 4.1 Instance Lifecycle

While the PE executes an instance, it goes through a number of states (see Fig. 3). Reaching a state also results in sending an event through the ■ data stream interface for all external services subscribed to the **state** topic.

**Ready** is the state that an instance is in, immediately after it is created. Instances in CPEE.org are not created with an initial model, but empty. Any UI allowing to instantiate a model as an instance, or any functionality instantiating a sub-process instance, therefore in the next step has to load a process model (through the ● control interface, which in turn triggers events being sent out through the ■ data stream interface). In ready state (a stopped state), changes to all aspects of an instance are possible: the instance model (description), data elements, endpoints, attributes, as well as the position in the instance model (description) that the execution should start from.

From there (1) a UI managing the instance, or (2) a functionality instantiating a sub-process instance can trigger a transition to state:

- **Running:** The instance is executed, activities are enacted.
- **Abandoned:** A manually set state (without proper execution) signifying that the instance is no longer able to run. For example, external services connected through the ■ data stream interface might have prohibited the proper loading of a process model into the instance, thus rendering the instance unusable. This state is final, and can not be left. No further changes to the instance are allowed.



**Fig. 3.** CPEE.org Instance Execution Lifecycle

If the execution of a process instance is successful (without an error occurring), the instance will transition to state **Finished**. This state is final, and can not be left. No further changes to the instance are allowed. The state finished can not be voted on, and can not be set through the  control interface.

If an error occurs during the execution of a process instance, the instance transitions to the state **Stopping**. This set can also be triggered for running instances at any time through the  control interface, and can be voted on by external interfaces through subscriptions to the  data stream interface.

The state stopping is an intermediate state to give functionalities the chance to go into a consistent state. Synchronous activities in parallel branches need still be able to collect responses from invoked functionalities. As soon as all functionalities have successfully returned values the instance state will transition to **Stopped**. Asynchronous activities do not contribute to delays. In stopped state callbacks will be suspended, when the state changes back to **Running**, callbacks are again accepted for an activity. Thus, (1) synchronous activities have to return before stopped state, (2) asynchronous activities can be suspended.

The **Purged** state is only reachable from **Running** and **Abandoned**. While for all other states, the instance can be inspected through the PE, after purging only logs created through the  data stream interface continue to exist.

## 4.2 Activity Lifecycle

Whenever the PE enacts an activity, the activity enactment transitions through a set of states (see Fig. 4), which also results in events being sent to external services subscribed to the **activity** topic.

**Syncing Before** and **Syncing After** are votes, thus external services can prohibit or delay the enactment of an activity (cmp. [7]). Both of these states are not part of the formal enactment of the activity but signify before and after enactment.

The **Calling** state is the first state of the enactment. It signals that input data is sent to the functionality implementing a certain task as part of the enactment

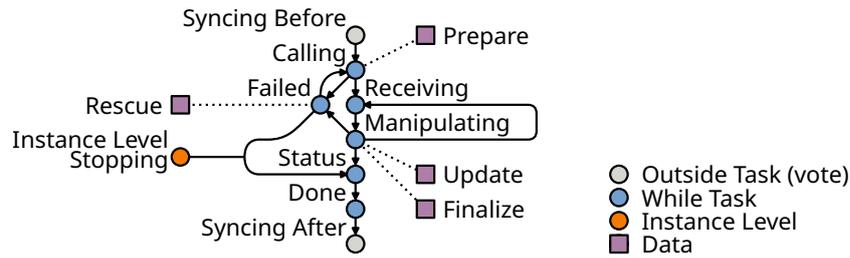


Fig. 4. CPEE.org Activity Lifecycle

of an activity. Before actually invoking functionality a **Prepare** script can be used to prepare the input data. Changes to the instance context (dataelements) made in this script are not permanent and only exist in the scope of a certain activity.

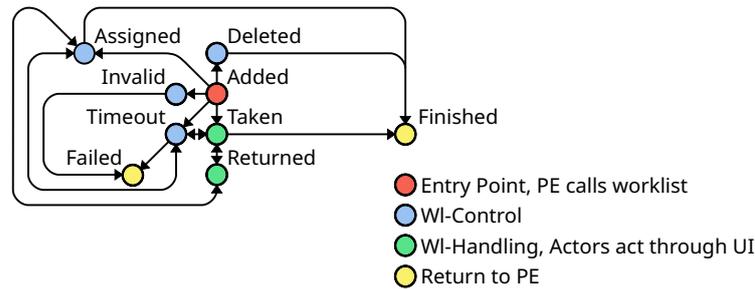
It the functionality responds, and data is received, the activity transitions to the **Receiving** state. Receiving data can happen as part of a synchronous or asynchronous interaction between an activity and the functionality it invokes. Depending on the amount of data the receiving phase takes a certain amount of time. For synchronous or asynchronous then the state transitions to **Manipulating**, so either **Finalize** or **Update** scripts are invoked. **Update** is called in case of an asynchronous update interaction (see above), **Finalize** is called in all other cases. Both scripts have full access to the received data, as well as to the instance context (dataelements) and can modify it permanently. In case of the asynchronous update pattern the state may again switch to **Receiving**.

If **Calling** or **Manipulating** fails (either by a functionality not available, or a response signifying some errors, or a update/finalize script having a syntax error), the activity transitions into the **Failed** state. In **Failed** state a script **Rescue** can clean up the instance context (dataelements), or set a special process status to tell the PE if it should retry invoking (**Calling**) the functionality, just ignore the error, or transition the instance to state **Stopping**. The activity then transitions to state **Status**, and subsequently to **Done**.

### 4.3 Task Lifecycle

Each task may have its own internal lifecycle, implemented in the functionality invoke by an activity. This lifecycle can be either hidden from the PE, or made transparent through a **CPEE-EVENT** response from the functionality through the **Δ** operation interface (see above). If the internal lifecycle is exposed, it will be sent out through the **■** data stream interface to all external services subscribed to the topic **task**.

Fig. 5 depicts a potential lifecycle for a human task, although other human tasks might implement different lifecycles. The human task in this particular case is



**Fig. 5.** CPEE.org Worklist Lifecycle

implemented as a worklist functionality. Each enacted activity can pass information about a task to the worklist functionality, which then coordinates humans to work on the tasks stemming from different activities in parallel branches, and different instances.

The lifecycle depicted in Fig. 5, depicts the possible states of one task. Whenever the worklist functionality is invoked as part of the enactment of an activity, the **Added** state is reached. As part of the internal functionality of the worklist the **Deleted** intermediate state might be triggered (e.g., when the task is a duplicate), leading to the state **Finished**.

Alternatively the **Invalid** state might be reached, e.g., if there is no suitable human worker being able to work on the task (e.g. because all workers are unavailable due to illness), leading to a **Failed** state.

Another possibility is triggering of state **Timeout** if a supplied deadline has passed, leading again to the **Failed** state.

The **Assigned** state is a special state that can be reached for certain classes of worklists that automatically assign tasks to humans:

- Round Robin worklist: work is assigned to a set of humans (e.g., sharing a common role) in round robin fashion. The first task is assigned to the first human, the second task to the second human, and so on. When all humans have a task, the first human is again assigned a task.
- Workload worklists: a random human belonging to a group (e.g., sharing a common role) with the lowest number of tasks is assigned a task.
- Skill based worklists: the human with the best set of skills matching the task description is assigned the task.

This state can be reached from **Added**, as well as **Timeout** (e.g., when a deadline is passed, the task is reassigned to a different human) states. This state can result in **Finished** state. Furthermore humans can signal that they can not do the task resulting in the state **Returned**. From there the task can be reassigned to a different human, resulting again in state **Assigned**.

An altogether different class of worklist is described by the remaining states. The **Taken** state can only occur in a worklist where tasks are not automatically assigned, but instead actively reserved by a human from a list of available tasks. Each task is typically visible to a group of humans sharing a common role. Taken tasks are no longer visible to other humans in that group. Taken tasks can either be **Finished**, **Returned** to the list of tasks for other humans in the common group to be reserved, or automatically **Assigned** to a human as described above.

**Failed** and **Finished** are the two final states reachable for a task. **Taken** and **Returned** are states triggered by human action, while all other states are typically the result of worklist internal mechanisms.

Activities invoke Worklists in an asynchronous manner, final responses occur when **Failed** and **Finished** states are reached. All other state changes might lead to intermediate responses (asynchronous update, see above), and thus to events sent to all external services subscribed through the  data stream interface.

## 5 Components

In order to assemble a service-oriented BPM like CPEE.org, the interfaces presented above can be used to create and connect a set of components, allowing for managing and operating a highly-scalable system.

### 5.1 Control Interface

Connected to the control interface are four main components.

The **Process Design UI (PDUI)** creates process models, by using the task repository (simple list of available endpoints), as shown in Fig. 7. The PDUI is a simple HTML/JavaScript SPA (single-page-application). It is connected to the  **control interface** to allow for testing the models on-the fly (by creating new instances). Whenever process models reach a certain maturity, they can be saved in the **Process Repository (PR)**. The PR, just acts as a storage front-end which versions each saved process model in arbitrary GIT repositories, which is important to comprehend changes, and cooperative work on models. Versions are created whenever a user saves the model into the PR (see Fig. 7, “Save” top left).

The stored and versioned process models can be managed through the **Process Lifecycle Management UI (PLMUI)** [6]. The PLMUI allows to manage the lifecycle of models. Each model can be either a graphical design draft (no endpoints), under development (not yet fully functional and tested), in production, or at its EOL in the archive (see Fig. 8). Each model, or each folder of models can be shifted between these four lifecycle stages, in the top right of Fig. 8, it

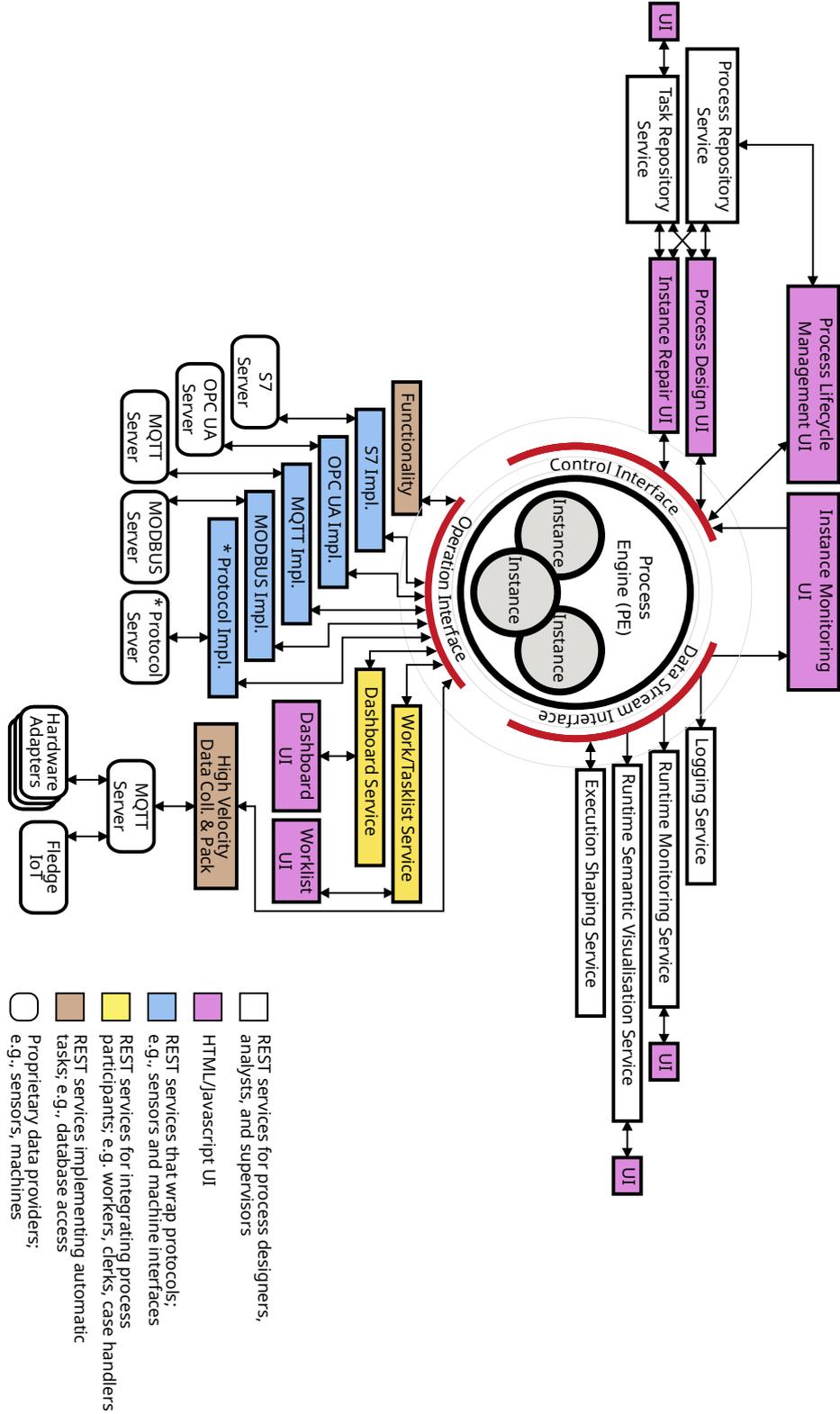


Fig. 6. CPEE.org Components And How They Use The Interfaces

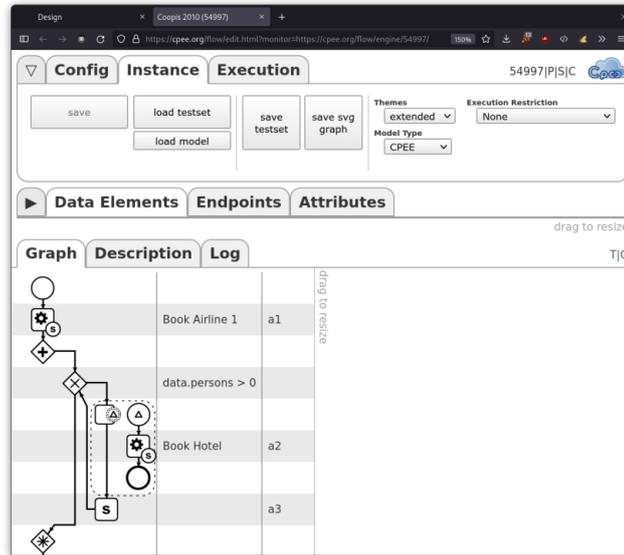


Fig. 7. CPEE.org Process Design UI

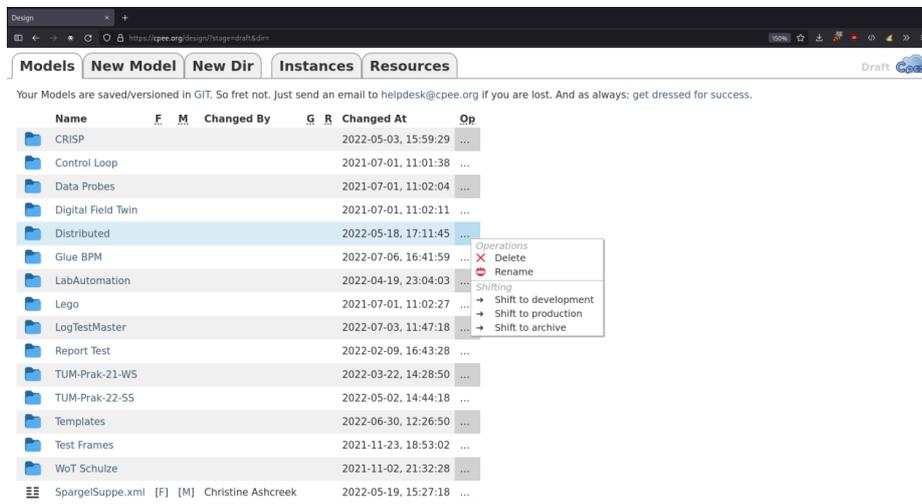
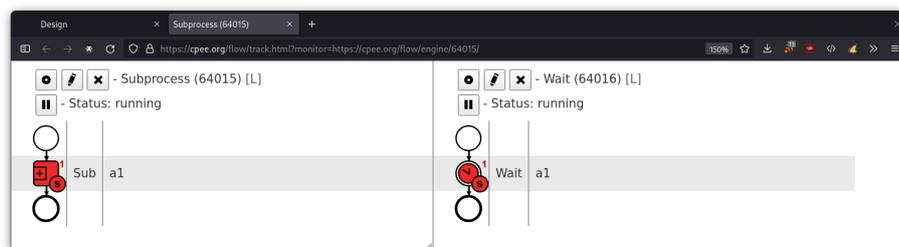


Fig. 8. CPEE.org Shifting Between Lifecycles

is possible to switch which lifecycle is currently displayed. While these four lifecycle stages are typical in Software Engineering, it is possible to configure the PDUI for additional lifecycle stages to match different, more extensive, or more basic development styles.

The PDUI (see Fig. 7) itself is designed as a cooperative editor, so when multiple people work on the same model, all edits are directly shown in all browsers currently viewing the model. This cooperative editing is realized through a SSE (server side events) subscription the  data stream interface (connection not shown in Fig. 6 for simplicity).

The **Instance Repair UI (IRUI)**, is very similar to the PDUI. It works on single instances, which are in state stopped (see Sec. 4.1 “Instance Lifecycle” above). Firstly, whenever changes are made, a user has the option to save the changes to the process model, for later instances of the process to include the fix. Secondly, the user can also apply the fixes to other running instances, which have not yet reached progressed to the point of the fix.



**Fig. 9.** CPEE.org Instance Monitoring UI

The **Instance Monitoring UI (IMUI)** (see Fig. 9) shows a life view of how instances are executed (red task is currently executed task). If an instance spawns one or many sub-process, they are shown to the left of the instance. For each instance, the IMUI offers controls to change the state (e.g. stop, start), change to edit mode (i.e., IRUI), or hide an instance from view. The IRUI is again an HTML/JavaScript SPA, which is subscribed to the  data stream interface, to receive information about which task is currently executed (red task), and the current state of each instance. It is also subscribed to the task topic to receive instantiation events, in order to show sub-processes.

## 5.2 Data Stream Interface

Exclusively connected to the  data stream interface are four main components.

The **Logging Service** has no UI. Its purpose is to subscribe to a set of events, in order to store an XES file on disk. The XES files are linked in the IRUI (see Fig. 7, Log UI Element, in the main/bottom right).

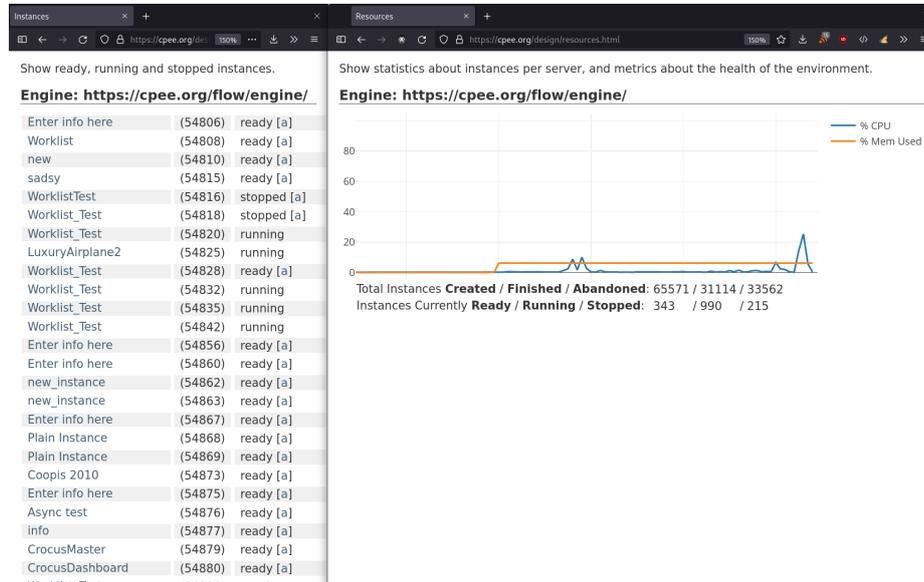


Fig. 10. CPEE.org Runtime Monitoring

The **Runtime Monitoring Service + UI (RMUI)** (see Fig. 10) consume events from the  data stream interface to provide information:

- Running Instances, their state (running, ready, stopped), as well as the memory usage per instance (left)
- Statistics about the overall memory and CPU usage (right).
- Statistics about total instances, as well as currently active instances (right).

Please note that the URL of the engine is very prominently shown, as the RMUI can be subscribed to multiple engines, which either together format a load-balanced cluster of engines or are unrelated. The functionality shown in Fig. 10 is achieved by the following subscription:

- topic **state**, event **change**: monitor the creation of instances (ready), as well as their full lifecycle as described in the Sec. 4.1.
- topic **task**, event **instantiation**: monitor the creation of sub-process instances. While **state/change** only provides information about the existence of an instance, this adds information about their parent/child relationship.

- topic **status**, event **resource\_utilization**: monitor the memory and CPU usage for instances.

A **Runtime Semantic Visualization Service + UI (RSVUI)**, is in contrast to the RMUI intended to subscribe mostly to topic **dataelements**, to monitor the data-flow in process instances, and topic **activity** to monitor the duration of individual tasks. An RSVUI is a custom **Key Performance Indicator (KPI)** monitoring and visualization service. For example, when a service is about production of parts, the number of parts, cycle times (time to produce one part, which might be the result of multiple activities enacted in a loop) or overall equipment efficiency (OEE) can be displayed. This can be realized in two ways:

- Bad solution: write a custom service, the hard-codes the meaning of certain data-element and tasks, in order to find and display the KPIs. Whenever the process model changes, it has to be checked if the RSVUI has to be changed as well, as it might breach when new activities are added, or the data flow changes.
- Good solution: annotate the BPMN with semantic information about how to extract the KPIs. Thus the RSVUI will be more generic and can, whenever events are received, inspect the corresponding BPMN for data extraction, transformation and display (ETD) information.

Both solutions can be observed in practical applications. While the first solution is sometimes preferred for less implementation overhead, the second solution is always better given that a suitably powerful semantic annotation, mechanism exists. For CPEE.org various aspect of such a BPMN extension are still subject of on-going research [1], for other BPMN editors, such research to the best of our knowledge is not easily possible or foreseen.

An **Execution Shaping Service (ES)** is a special component that subscribes to arbitrary events and votes from the **■** data stream interface, and **enacts actions through votes** as described in Sec. 5.2 “Data Stream Interface”. Examples for such services are:

- Runtime compliance checking: directly reacting when compliance violations are detected. Simple cases might include stopping an instance, and notifying responsible actors, complex cases might include the automatic modification of responsible actors in the instance process model to fix the compliance violation.
- Self-healing: in case of errors occurring with endpoints (e.g., machines), fix the instance by changing endpoints, or triggering compensation.
- Load-Balancing: at runtime change endpoints to select resources (endpoints) with the least workload.

Many other applications exist, as for RSVUI these applications might be very domain specific and might require additional information in the process model to

keep the ES generic enough to not break its functionality when process models evolve.

### 5.3 Operation Interface

Components using the  $\blacktriangle$  operation interface should not use any other interfaces, as this will have negative effects on the security of the overall system. Separation of the enactment of activities, or rather the functionalities they are linked to, will guarantee that everything can be properly tracked and observed, and no behaviour can be hidden.

For CPEE.org currently all components connected to the operation interface are realized as REST services, although this is not a requirement, as the engine supports pluggable  $\blacktriangle$  operation interfaces that could support arbitrary protocols.

For CPEE.org all REST services follow the synchronous, asynchronous, or asynchronous update patterns, as described in Sec. 5.3.

The services connected to the  $\blacktriangle$  operation interface fall into four generic groups:

- **Basic Functionality:** unspecified functionalities that perform automatic tasks such as extracting data from a database, or extracting data from CRM. They are black boxes, and implement a specific interface.
- **Protocol Proxies:** a class of services that wrap custom, often proprietary protocols. Currently CPEE.org supports S7, OPCUA, MQTT and MODBUS, which proved sufficient for many industrial applications. As these protocols might have very different communication patterns which might solely rely on pushing messages, the proxy service mostly use the asynchronous pattern to interface with the process engine.
- **User Integration:** User integration is again a form of proxy service, but with the goal of integrating users into an enacted. For this a Work-/Tasklist or Dashboard Service has to utilize the HTTP headers as described in Sec. 5.3 “Operation Interface”, store callback information and parameters in provided parameters, and tell the process engine that asynchronous pattern is to be used. A separate UI, solely used by Process Participants then utilized the provided parameters to present an UI, as discussed above. CPEE.org provides both, a worklist as well as a dashboard component to build interactive user interfaces. While in office automation worklists are more common, on the shop-floor dashboards are more prevalent.
- **High velocity data collection and packaging:** In IoT environments such as shop-floors, typically two kinds of components exist: Sensors and Actuators. Actuators can be triggered to start some sort of operation. They are typically directly represented as activities in process models, and typically return a result describing success/error of an actuation. Sensors on the other hand observe various properties of the shop-floor. They might range from

something as simple as measuring temperature and humidity, to monitoring all conditions inside a machine, such as the power consumption of individual motors, or the position of various axes of a lathe. There are two different scenarios that might occur:

- Sensor information is related to one particular activity. E.g., for an activity “Machine Part” all sensor information regarding this machining operation can be collected, and directly attached to the task.
- Sensor information is continuous and not related to one particular activity, but rather to a group of activities, an instance, or even a group of instances. Thus the information can not be attached to single activities but to a higher plane of structure.

**User Integration** is the most prominent use-case for utilizing the  $\blacktriangle$  operation interface. Traditionally work-/tasklists were integrated into monolithic BPMs, instead of being loosely coupled with it through a common interface for all services. This can be also seen when looking, e.g., at the XES standard for storing log information. Its lifecycle extension is mixing the lifecycle for activities (see above) with the Lifecycle for tasks (see above), because in the traditional view there was no separation between these. The traditional view does not consider the different/specialized work-/tasklists could have different and much more fine-grained lifecycles.

Traditional work-/tasklist (being integrated into the BPM) also assume that they have access to the full process context, i.e. all data elements, attributes and other internal information. For CPEE.org, with its focus on modularity, loose coupling, and strict separation of concerns, this is not true. Utilizing the  $\blacktriangle$  operation interface induces that all information required for a work-/tasklist to do its job is passed to the functionality that implements it. Information required by a tasklist might include:

- (A) Which user/role is to work on a task.
- (B) Which organization structure is to be used to select users based on roles.
- (C) UI/Form which should be shown when working on the task.
- (D) Task specific information that has to be known to the user working on the task in order to be able to do it.
- (E) Deadlines.

A **work-/tasklist** should return (at least) at least (1) the result of the work (e.g., a document or success notification), and (2) which user(s) worked on the task in order to be able to do compliance checking.

It is not wise to handle Separation of Duty (SoD) / Binding of Duty (BoD) internally in a work-/tasklist, but to escalate it to the level of the process. This should be realized by treating the passed information (a) (see enumeration above) as follows:

- Separation of Duty (SoD): in addition to the role (list of people who can do a task), pass information about a user / users from this list, who are NOT allowed to do this task. The not-allowed-users can be collected from the return of previous calls to work-/tasklists which are part of the SoD logic.
- Binding of Duty (BoD): instead of passing a role (list of people who can do a task), only the user that preciously worked on tasks which are part of the BoD logic.

By implementing this on the level of the process, two main advantages:

- Simple Compliance Checking: Compliance checking can be realized at the process level, instead of being required to access information from private logs of the work-/tasklist service.
- Centralized Configuration: All configuration is part of the process model, instead of being hidden inside functionality. Different work-/tasklists can be mixed, without the requirement of accessing common configuration information.

**Dashboards** are different from work-/tasklists, as they are either (1) read-only, or (2) bound to a physical location. Because of this, they might be access restricted, but there is no need to specify user/role or organizational information. They also typically only show one thing. So when triggering a dashboard either (1) the information shown is replaced, or (2) added to be shown simultaneously in an other part of the dashboard.

High velocity data collection and packaging realizes a special service which can be used in two ways:

- A process engine (PE) can ask for packaged data from not one or a group of sensors. In this case the process model contains instructions to explicitly collect sensor information.
- A process engine (PE) implements that in parallel to the execution of instances or the enactment of activities sensor information is collection. In this case the process models contains additional information to collect information while an instance is executed or while a group of activities is enacted.

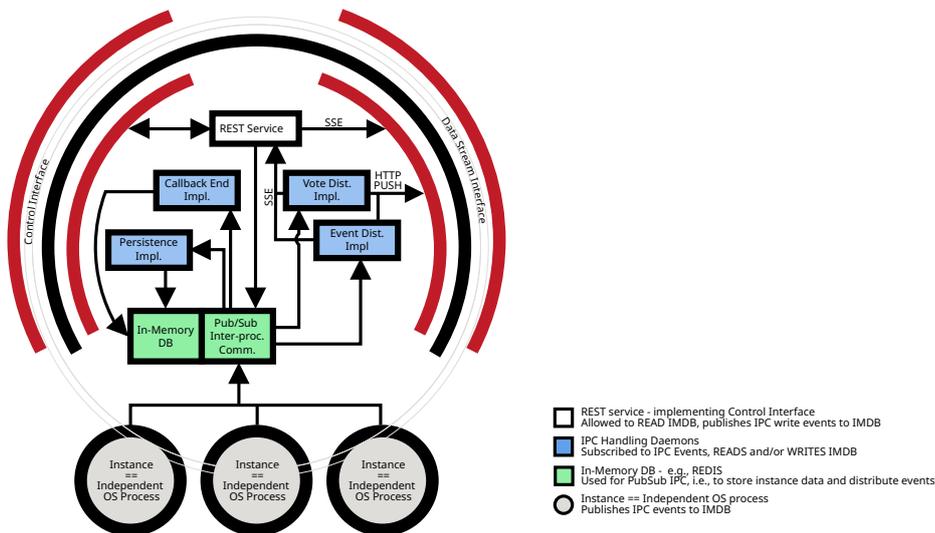
CPEE.org implements both mechanisms, both through normal BPMN and BPMN IoT extensions.

## 6 Highly Scalable Architecture

CPEE.org is not only modular by providing a set of interfaces to the outside, but it is highly scalable by internally also being based on a set of services. This allows to distribute one CPEE.org over multiple nodes (scale-out architecture) in a number of different communications, best suiting the needs of a wide range

of application scenarios. As depicted in Fig. 11, the first important part is: **Each instance is a separate service**, communicating with the rest of the PE through an IPC mechanisms with Publish/Subscribe functionality, connected to an in-memory data-base to store the internal state of an instance execution. This introduces the following properties:

- Instances can be deployed to different nodes for maximum scalability.
- Instances can be deployed into separate containers for improved security properties.
- Instances are managed by the underlying OS as processes, meaning any number of CPU cores is transparently used.
- Instances can be monitored through standard monitoring facilities, their CPU and memory usage is always separately available.
- Instances can be restricted with separate CPU / Memory quotas through standard OS mechanisms.



**Fig. 11.** CPEE.org Highly Scalable Service Oriented Architecture - Internal Message Routing

**Each instance implements the  $\Delta$  operation interface.** Each instance can be realized in two different ways:

- By having an interpreter read and execute the statements in a BPMN.
- By translating/transpiling the BPMN into a native language and then compiling/executing the result.

CPEE.org uses the second mechanism as it provides higher performance and lower overhead. In general it has to be noted that realizing instances as standalone services has also some drawbacks:

- The necessity of using IPC introduces some serious overhead, when compared to having a multitude of instances being executed (interpreted) inside a monolithic process engine. We think the possibility of scaling mitigates this disadvantage.
- The memory overhead can be considerable, as each instance potentially has to carry and run its own BPMN interpreter. CPEE.org avoids this by employing the transpilation mechanism.

CPEE.org employs Redis<sup>4</sup> as the in-memory database. Any change to data elements, endpoints, or attributes, has to be made available to the Pub/Sub mechanism by the instance. Furthermore information about which task is currently executed (including lifecycle information) has to be made available as well. In fact, all information available as an event, as described in Sec. 4) has to be constantly sent while each instance is running.

The remaining components inside the PE are subscribed to this event-stream and act on it as follows:

- The **Persistence Implementation (PI)** saves the information in the in-memory database.
- The **Event Distribution Implementation (EDI)** directly sends relevant information to subscribed services, according to the logic described in Sec. 5.2. While HTTP based push messages can be distributed (fire and forget) directly to subscribers, Server Sent Event (SSE) subscribers have to be handled differently (see below).
- The **Vote Distribution Implementation (VDI)** does the same as the EDI but for votes. This component also handles the answers to votes, which can only arrive through HTTP (see below).
- The **Callback End Implementation (CEI)** which is responsible for cleaning up the database after a response to a vote or an asynchronous call.

The final and most important service is **REST Service (RS)** which implements the **● control interface**. All input to the **● control interface** again is sent to the Pub/Sub mechanism, and thus is distributed to the internal components. Each **process instance** is subscribed to certain events as well. For example a stopping state change, triggered through the **● control interface**, has to be received by the respective instance, which then has to stop running: (1) it has to wait for all synchronous calls through the **▲ operation interface** to finish, (2) has to announce state stopped, and (3) has to exit (on the OS level, thus releasing all memory).

<sup>4</sup> <https://redis.io/>

While EDI and VDI actively distribute votes and events over HTTP, the possibility to subscribe to this through SSE, brings the necessity for the REST service RS to deliver events as well. Thus the EDI and VDI send special IPC messages, to which the RS is subscribed, which are then sent through SSE. This allows for example the **Instance Monitoring UI** component (see section above), which is just HTML and JavaScript to receive the necessary information to update its UI.

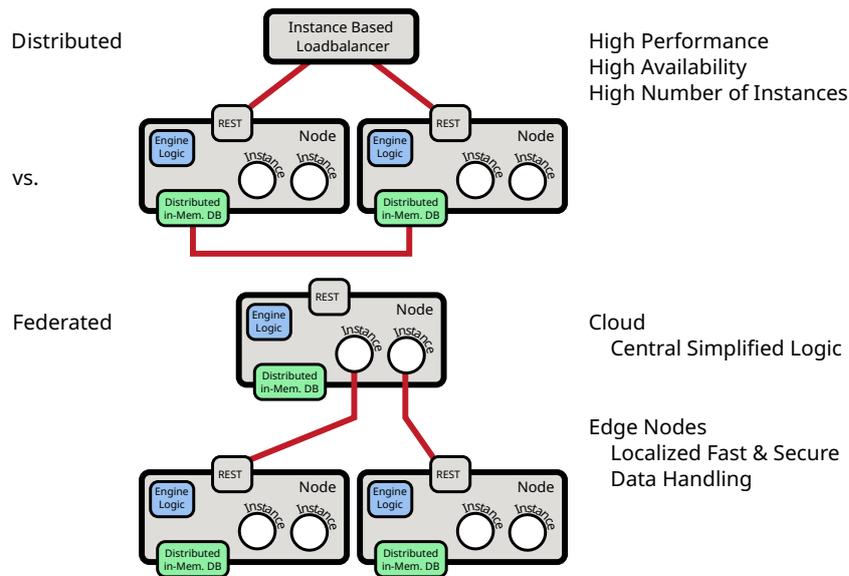


Fig. 12. CPEE.org Distributed Vs. Federated

With these internal services, the PE can be set up in many possible ways, some of which are depicted in Fig. 12. In the **Distributed Scenario**:

- Each node has its own Distributed in-memory database, but they are operating as one cluster. This is for example supported by Redis as used in CPEE.org, but also supported by others.
- Instances run on nodes.
- All other services, including the REST service are available on each node.
- A centralized load-balancer distributes **control interface** HTTP traffic based on the instance id. E.g., for two nodes all even instances are hosted on node one, all odd instances on node 2.

Of course different more complicated load-balancing mechanisms can be realized easily. Another possibility is to not host the instances on the same node but to

distribute them to separate nodes. Separate nodes for EDI/VDI, which are easily the most taxing services, are possible as well.

Another important scenario is the **Federated Scenario**. For IoT/Edge use-cases different Process Engines (PE) can run on different nodes. As the creation of sub-process instances is realized through a special **▲ operation interface** component, federated PEs can be used like normal services.

## 7 Conclusion

CPEE.org realizes a Process Engine, which goes beyond the state-of-the of both, industrially and scientifically available offerings. Its core and many components are open-source<sup>5</sup>, actively maintained and constantly extended. Its no-compromise architecture makes it particularly well suited for taxing industrial applications. It is also well suited for University teaching, due to its robustness (separate instances) and security (instances runnable in containers).

Due to its modularity, while maintaining three simple and streamlined interfaces, it is very well suited for research. All aspects of BPMN can be customized through external services, without the necessity of learning any specific technologies or programming languages.

While more than 15 years old, it maintains a healthy community of developers and users, both from industry [8] and academia.

## References

1. Ehrendorfer, M., Mangler, J., Rinderle-Ma, S.: Sensor Data Stream Selection and Aggregation for the Ex Post Discovery of Impact Factors on Process Outcomes. In: International Conference on Advanced Information Systems Engineering. vol. 424, pp. 29–37. Springer, Melbourne, Australia - online (2021). [https://doi.org/10.1007/978-3-030-79108-7\\_4](https://doi.org/10.1007/978-3-030-79108-7_4), [https://link.springer.com/chapter/10.1007%2F978-3-030-79108-7\\_4](https://link.springer.com/chapter/10.1007%2F978-3-030-79108-7_4)
2. Mangler, J., Beran, P., Schikuta, E.: On the Origin of Services Using RIDDL for Description, Evolution and Composition of RESTful Services. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CC-Grid). pp. 505–508 (May 2010). <https://doi.org/10.1109/CCGRID.2010.126>
3. Mangler, J., Schikuta, E., Witzany, C.: Quo Vadis Interface Definition Languages? Towards an Interface Definition Language for RESTful Services. In: Service-Oriented Computing and Applications (SOCA'09), International Conference on. IEEE Computer Society, Taipeh, Taiwan (2009)
4. Mangler, J., Rinderle-Ma, S.: CPEE - Cloud Process Execution Engine. In: Limonad, L., Weber, B. (eds.) Proceedings of the BPM Demo Sessions 2014 Colocated with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014. CEUR Workshop Proceedings, vol. 1295, p. 51. CEUR-WS.org (2014), <http://ceur-ws.org/Vol-1295/paper22.pdf>

<sup>5</sup> <https://github.com/etm/>

5. Mangler, J., Stuermer, G., Schikuta, E.: Cloud process execution engine-evaluation of the core concepts. arXiv preprint arXiv:1003.3330 (2010)
6. Mangler, J., Pauker, F., Rinderle-Ma, S., Ehrendorfer, M.: centurio.work - Industry 4.0 integration assessment and evolution. In: Brocke, J.v., Mendling, J., Rosemann, M. (eds.) Proceedings of the Industry Forum at BPM 2019 co-located with 17th International Conference on Business Process Management (BPM 2019), Vienna, Austria, September 1-6, 2019. CEUR Workshop Proceedings, vol. 2428, pp. 106–117. CEUR-WS.org (2019), <http://ceur-ws.org/Vol-2428/paper10.pdf>
7. Mangler, J., Rinderle-Ma, S.: Rule-Based Synchronization of Process Activities. In: Hofreiter, B., Dubois, E., Lin, K.J., Setzer, T., Godart, C., Proper, E., Bodenstaff, L. (eds.) 13th IEEE Conference on Commerce and Enterprise Computing, CEC 2011, Luxembourg-Kirchberg, Luxembourg, September 5-7, 2011. pp. 121–128. IEEE Computer Society (2011). <https://doi.org/10.1109/CEC.2011.23>, <https://doi.org/10.1109/CEC.2011.23>
8. Pauker, F., Mangler, J., Rinderle-Ma, S., Pollak, C.: centurio.work - Modular Secure Manufacturing Orchestration. In: Aalst, W.M.P.v.d., Casati, F., Conforti, R., Leoni, M.d., Dumas, M., Kumar, A., Mendling, J., Nepal, S., Pentland, B.T., Weber, B. (eds.) Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018. CEUR Workshop Proceedings, vol. 2196, pp. 164–171. CEUR-WS.org (2018), [http://ceur-ws.org/Vol-2196/BPM\\_2018\\_paper\\_33.pdf](http://ceur-ws.org/Vol-2196/BPM_2018_paper_33.pdf)
9. Stertz, F., Mangler, J., Rinderle-Ma, S.: Temporal Conformance Checking at Runtime based on Time-infused Process Models. CoRR **abs/2008.07262** (2020), <https://arxiv.org/abs/2008.07262>, eprint: 2008.07262
10. Stertz, F., Rinderle-Ma, S., Mangler, J.: Analyzing Process Concept Drifts Based on Sensor Event Streams During Runtime. In: Fahland, D., Ghidini, C., Becker, J., Dumas, M. (eds.) Business Process Management - 18th International Conference, BPM 2020, Seville, Spain, September 13-18, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12168, pp. 202–219. Springer (2020). [https://doi.org/10.1007/978-3-030-58666-9\\_12](https://doi.org/10.1007/978-3-030-58666-9_12), [https://doi.org/10.1007/978-3-030-58666-9\\_12](https://doi.org/10.1007/978-3-030-58666-9_12)