Scalable SaaS-Based Process Customization with *CaseWalls*

Yu-Jen John Sun(✉), Moshe Chai Barukh, Boualem Benatallah, and Seyed-Mehdi-Reza Beheshti

School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{johns,mosheb,boualem,sbeheshti}@cse.unsw.edu.au

Abstract. The rising popularity of SaaS allows individuals and enterprises to leverage various services (e.g. Dropbox, Github, GDrive and Yammer) for everyday processes. However, these disparate services do not in general communicate with each other, rather used in an ad-hoc manner with little or no customizable process support. This inevitably leads to "shadow processes", often only informally managed by e-mail or the like. In this paper, we propose a framework to simplify the integration of disparate services and effectively build customized processes. The implementation of the proposed techniques includes an agile services integration platform, called: *CaseWalls*. We provide a knowledge-based event-bus for unified interactions between disparate services, while allowing process participants to interact and collaborate on relevant cases.

1 Introduction

Traditional structured process-based systems increasingly prove too rigid amidst today's fast-paced and knowledge-intensive environments. A large portion of processes, commonly described as "unstructured" or "semi-structured" processes, cannot be pre-planned and likely depend upon human-interpretation. On the other hand, there has been a plethora of apps to support everyday tasks with enhanced collaboration. For example, *Software-as-a-Service (SaaS)* based tools such as: (i) *Dropbox* to store and share files; (ii) *Pivotal tracker* to manage tasks and projects; and (iii) *Google Drive* to edit and collaborate. Workers often need to access, analyze, as well as integrate data from various such cloud data services.

Albeit, there are crucial gaps in the SaaS-enabled endeavor: The large number of available services do not easily communicate with each other - often employed ad-hoc. Moreover, such ready-made services implies conforming to a fixed set of embedded features allowing little or no room for customization. Alternatively, even if a collection of such services are used for different portions of tasks, this inevitably leads to "shadow processes", where synchronization between such services is handled in an ad-hoc manner (e.g., actions are often accomplished in a number of non-traceable steps via manual tasks, such as email or the like).

At the same time, the intent of the SOA-based approach was to simplify service integration via APIs. However, this was met by the inherent need to

A. Barros et al. (Eds.): ICSOC 2015, LNCS 9435, pp. 218–233, 2015.
DOI: 10.1007/978-3-662-48616-0_14

understand various low-level APIs, leading to inflexible and costly programming environments, requiring multiple and continuous patches. To counteract this, advances in process composition languages emerged (e.g. BPEL), along with Mashup environments (e.g. Yahoo! Pipes). Albeit, these environments rarely provide productivity support tools akin to modern IDEs (e.g. code search and discovery, ease of reuse, debugging, code generation). Moreover, they suffer from the lack of agility and cannot support run-time changes. We thus call for an exciting new change: where advanced techniques for *simple*, *declarative*, *flexible exploration* and *manipulation* of multiple services in large scale and dynamic environments, are needed. We argue, the ubiquity of process and services will have little value if users cannot use, share and reuse them simply.

To address the above challenges, in this paper we propose a framework to *simplify* service-integration and effectively build customized processes. Central is the notion of *Case Knowledge Graph (CKG)*, where common services-related low-level logic can be abstracted, incrementally *shared* and thereby *reused* by developers. We organize knowledge into various dimensions: *API*s, *Resources*, *Events* and *Tasks*. By identifying entities (i.e. attributes and relationships, along with their specialization), a novel foundation is introduced to accumulate dispersed case knowledge in a structured manner. This offers a unified representation, manipulation and reuse of case knowledge to empower simplified SaaS-enabled process customization. Empowered by this knowledge graph, we provide a novel case customization and deployment platform, called *CaseWalls*, which enables:

- Professional process developers to incrementally create modular collections of tasks - reusable and customizable process fragments (referred to as a *"case"*). E.g. Create an issue on a project management service; upload a file into a document management service; send an email when a co-worker upload new version of a file; post videos and photos into social media services, etc.
- A simple, declarative yet powerful language that allow case-workers to search existing tasks and compose into customized definitions. Composite tasks are abstracted as reusable cases in the CKG for further reuse. The tasks search component uses a *"context"* to describe the task *"intent"* and *"objective"* (e.g., upload a file, create an issue). Thus, using the business scenario to query the CKG can return tasks that are appropriate for the given context.
- An event/activity "wall" to inform case-works about task progress; together with a simple and declarative language to enable such participants to uniformly and collectively react, interact and collaborate on relevant case.
- The above is supported by a unified, knowledge-based event-bus for case orchestration. The knowledge required at runtime to orchestrate cases (i.e., detecting events, executing tasks, invoking APIs) is automatically extracted from the defined case definitions and expressed as event-action case orchestration rules. (We thus reuse a rule-engine as our execution environment.)

The rest of this paper is organized as follows: In Sect. 2, we propose a unified case knowledge-graph and describe the constituent entities and relationships. In Sect. 3, we present a novel knowledge-driven and declarative case manipulation

language. In Sect. 4, we present our implementation; evaluation in Sect. 5; then, related-work and conclusions in Sect. 6.

2 Case-Knowledge Representation and Reuse

A knowledge-graph (KG) is an effective technique for taxonomy-based organization of concepts and relationships. For example, Google-KG[1] is a graph of popular *informational* concepts on the Web, such as "people, places and things". In our work, we apply a KG to curate (and thereby enable reuse) of service and process related programming concepts (i.e. *API*s, *Operations*, *Resources*, *Events*, *Tasks*) and their relationships. Formally, a KG is defined as an ordered pair $G = (V, E, A_v, A_e)$ where, V is the vertex set whose elements are the vertices (i.e. nodes or entities) of the graph; E is the edge set whose elements are the edges (or relationships) between vertices of the graph; A_v is a set of attributes that can be associated with any vertex in V; and A_e is a set of attributes that can be associated with any vertex in E.

Reference Scenario. For purpose of illustration, we describe a running example that we adopt throughout this paper (also further elaborated in Sect. 5). Consider the integration of a version-control and online source-code repository system *Github*, with a story-tracking system *Pivotal Tracker*. Often code traceability, collision and bug-repair require effective peer review and collaboration. The integration of these two tools would provide a powerful combination.

Figure 1 illustrates a canonical graph of possible relationships between entities, (where entities are structured data-objects with unique identifiers, and instance of entity-types, i.e. *API*s, *Operations*, *Resources*, *Events* and *Tasks*). Illustrating the above scenario, we may learn that Pivotal Tracker consists of a few *Operations* that manipulates Story and a `StoryFinished` Event that's

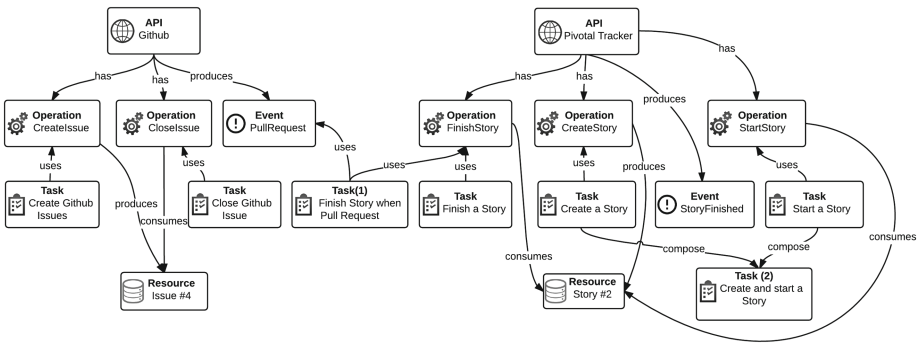

Fig. 1. A canonical graph of possible relationships between entities in the Knowledge Graph for case management systems.

[1] http://www.google.com/insidesearch/features/search/knowledge.html.

produced when a Story is finished. In addition, the *Automated Task(1)* will trigger the `FinishStory` *Operation* when a `PullRequest` Event from Github is received, providing a simplified work environment for the developer. In Pivotal Tracker, after a story is created, it has to be started manually afterwards to initiate the story. To simplify these steps, *Task (2)* can be used to shortcut this procedure. In the following, we define specific *entity-types* that are relevant:

API. Application Program Interface, represent the plethora of tools (e.g. storage, location, social-networking, etc.) exposed via Web-service interfaces (e.g. REST, WSDL). An API thereby encapsulates the *endpoint* and *operations*, (and other relevant information, e.g. OAuth protocol, refer [11]). We build upon our previous work [4], for a unified-structured approach of Web-APIs in the KB.

Operation represent the set of operations offered by a specific service (e.g. `CreateIssue` on Github, or `CreateStory` on Pivotal Tracker). Operations may *consume* or *produce* resource-types; and may also be used as *actions* within *tasks*.

Resource represent the input or output for an API. In this context, resources may have various granularities, from a large dataset to an entity representing issues in Github, stories in Pivotal Tracker, and even pdf files in Dropbox. To proficiently represent resources, we utilize JSON-Schema Draft v4.

Event is the record of an activity. Denoted E and consisting of attributes $\{T, R, \tau, D\}$, where T is the type of the event (e.g. Github provides 25 different event[2], such as *Push*, *Issue*, and *Fork*); R is the actor (i.e. person or device); τ is the timestamp; and D is a set of data elements recorded with the event (e.g. the task associated with the event). Events may be fine-grained (e.g. concrete events defined at the API level), or coarse-grained that capture a pattern relevant to a collection of resources (e.g., related stories in Pivotal Tracker).

Task represents the set of operations to achieve a defined goal, (ranging from a single to a combination of several API endpoints). Moreover, we propose the novel feature of *"context"*, which describes the *"intent"* and *"objective"*. This is especially useful to human process-designers or human-driven search engines. In this manner, operation endpoints may be mixed-and-matches between different APIs, to provide a more comprehensive *Task* that precisely targets a particular use-case. For example, the task `CreateIssue` may conform to different intents such as *report bugs*, *create pull requests* or *request feature*. Functionally, there are two task-*types*: *Automated* and *Manual*. The former assigned to an *Event* that may trigger the task, while the latter only triggered manually by an actor.

Tasks may also have sub-tasks, in order to help reduce complexity. For example, a `Code Commit` Task could be defined as a set of sub-tasks containing `GithubPush` and `GithubPullRequest` APIs. Utilizing Resources and Tasks nodes, we can identify potential service integration patterns, i.e.: If a Task T_A *produces* Resource R and Task T_B *consumes* Resource R, then Task T_A and T_B can be invoked in a sequence. For example, the Task `CreateIssue` *produces* an `Issue` and Task `EditIssue` *consumes* an `Issue`, therefore we can state that `EditIssue` can be invoked after `CreateIssue` on the same resource.

[2] https://developer.github.com/v3/activity/events/types/.

Relationship is denoted as $R = (E_1, E_2)$, which indicates a connection between entities E_1 and E_2. As illustrated, we have seven types of relationships: *"API has OPERATION", "TASK use OPERATION", "TASK use EVENT", "OPERATION consumes RESOURCE", "OPERATION produces RESOURCE", "TASK compose TASK", "OPERATION trigger EVENT"*. In addition, there is *"EVENT compose EVENT"* indicating that an event is complex, and is representative of some pattern composed of several other events.

3 Knowledge-Reuse-Driven and Declarative Case Definition Language

The notion of *Case* conceptualises a lightweight process (set of service interactions). A case is thus defined to consist of: *Tasks*, *People* and *Event*. *Tasks* indicate the services and the features needed in the interaction. *People* describes who have access to the Case, especially the *owner* who has the privilege to edit the case. Cases can contain both automated tasks. As well as manual tasks, when human discretion and thereby intervention is required. However, *Tasks* can also monitor *Events* (or patterns thereof) which may serve as notification to participants (e.g. perform some manual task). Moreover, as *Cases* themselves are represented as nodes which can be curated and reused in a modular manner. While the platform is exposed via a RESTful interface, we further propose a higher-level command-line Case Search, Definition and Interaction Language.

3.1 Knowledge-Reuse Language

Selecting the required tasks for a Case may not be trivial with an extensively populated knowledge-graph. We thus propose an effective search component (utilizing the index of the tasks objective and an iterative keyword search approach [10]). The closest matching tasks are thus recommended to the user, based on the objective tags described in Sect. 2. Albeit, the decision of whether a Task (or Sub-task) should be included finally relies on the Case designers' discretion (Fig. 2).

```
expression ::= "/" <op><keywords>
op         ::= "task"     #matches tasks against all possible related keywords
             | "resource" #tasks that are related to the resources matching the keywords
             | "input"    #tasks that consumes the resources matching the keywords
             | "output"   #tasks that produces the resources matching the keywords
             | "API"      #tasks that are directly related to the API specified
             | "event"    #tasks that are monitoring the events specified
             | "case"     #directly matches cases against all possible related keywords
keywords   ::= {<string>}  #set of keywords to perform the search
```

Fig. 2. Search Language Syntax

```
expression      ::= <new_case> | <extend>
new_case        ::= "CREATE CASE" <name><own>
                    [<shared>][<using service>][<monitor events>][<include tasks>]
extend          ::= "EXTEND CASE" <name><new name><own>
                    [<shared>] [<using service>][<monitor events>][<include tasks>]
own             ::= "OWNED BY" < user >
shared          ::= "SHARED WITH" < user > [{", " < user >}]
using_service   ::= "USING SERVICE"<service>[{","<service>}]
monitor_events  ::= "MONITOR EVENTS" < event > [{", " < event >}]
include_tasks   ::= "INCLUDE TASKS" < task > [{", " < task >}]
```

Fig. 3. Case Definition Language

3.2 Declarative Case Definition Language

Cases can be defined (or extended) using the language as defined below. The syntax contains governance policies over both people and services. For people, constructs such as OWNED BY (permission for editing/updating); SHARED WITH (permission for interacting). For services, the USING SERVICE construct indicates authorizations information - as the owner has to specify which authorization to share between the users of the Case. The MONITOR EVENTS construct details which events are to be monitored and notified to the participant users. While INCLUDE TASKS configures tasks related to the interaction (Fig. 3).

Figure 4 illustrates an example of a *GitHub Code Review Process with Pivotal Tracker*. Ordinarily, using Github alone, the `Lead Developer` may review code

```
CREATE CASE "CodeReview"
  OWNED BY "Project Manager"
  SHARE WITH "Project Manager", "Lead Developer"
  USING SERVICE "GitHub", "Pivotal Tracker"
  MONITOR EVENTS PullRequest,        #pull request has been 'received'
                 PRMerged,           #pull request has been 'merged'
                 ReviewFinished      #review has been completed
  INCLUDE ACTION MergePR,            #merge a pull-request on GitHub
                 CreateStory,        #create a story on pivotal tracker
                 FinishStory,        #mark a story finished on pivotal tracker
                 StartStory,         #start a story on pivotal tracker
                 DeliverStory,       #deliver a story on pivotal tracker
                 AcceptStory         #mark a story as accepted on pivotal tracker
  INCLUDE TASKS CreateReviewOnPR,    #invokes 'CreateStory' to create a new story to do review
                DeliverOnPR,         #invokes 'DeliverStory' when a pull-request is received
                MergeOnFinish        #invokes 'MergePR' when a review story is done
```

Fig. 4. CodeReview Case Definition Example

by requiring the `Engineers` to submit their code in forms of Pull Requests[3] and then review it on Github. However, if the project manager wish to monitor the review progress on Pivotal Tracker, they will have to manually create review tasks (stories) for lead engineer every time a review is needed. Using *CaseWalls*, we can simplify this process by defining a case and linking automated tasks to auto-create stories when receiving a pull request. Events can also be monitored, notification posted to interested participants and thereby manual tasks (e.g. upon completing the review, merge the code into upstream, and close review process) can also be accomplished.

3.3 Declarative Case Manipulation Language

Finally, we propose a interaction language to interact with the Case during execution. Interactions may be with both manual tasks (awaiting human intervention), as well as automated tasks (as in tapping into some 3rd-party service) (Fig. 5).

```
expression   ::=<op><params>
op           ::=< op task > | < op resource >
op_task      ::= "#" <task >
op_resource  ::= "@" <resource>
params       ::=<resource> |{<input> "=" <value>}
```

Fig. 5. Case Manipulation Langauge

Figure 6 illustrates *CaseWalls* for 3 participants in the *Code Review* process defined earlier. On the left, we see a set of notifications - this informs the participants what actions to take (if any). Actions might be interacting with some external software service (e.g. [P] Pivotal Tracker, and [G] GitHub]), or performing some manual task (i.e. [MT]). However, moreover behind the scenes automated tasks (i.e. [AT]) are also being performed as defined in the Case. It is thus apparent without *CaseWalls* all interactions would be done manually, with little or no flexibility. Not only do *CaseWalls* help automate certain tasks, it also automates the notification process - thus making it more simpler for participants to identify what needs to be done. Subsequently, the interaction language can be used to call upon manual tasks in a simple manner. As for implementation, the semantics of the language are translated into Rule-based expressions for the purpose of execution (refer to next section, at Sect. 4.4).

[3] https://help.github.com/articles/using-pull-requests/.

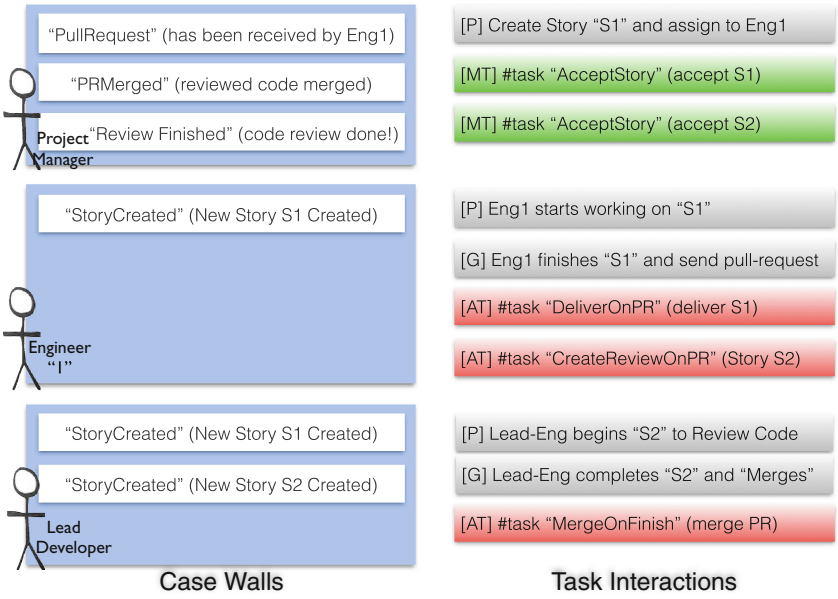

Fig. 6. CaseWalls with Illustration of Interactive Behavior

4 Implementation

4.1 Architecture

Figure 7 illustrates the system architecture and interaction of the main components of the *CaseWalls* platform. At the heart of the system is the *Knowledge-Graph (KG)* for Case-processes. It maintains the ontological relationships between key entities and facilitates task/case-based processes. Manipulation as well as effective searching of the case-based KG are conducted via the respective components (as shown). The *Event-Management* system: collects raw event-data from different services; and thereby, processes them using the patterns in the KG. We leverage our previous work for this, [4,5]. This feeds into the *Rule-Engine* which performs pattern-matching and can infer which actions to perform by calling upon the *Task Execution Engine*.

Overall, *CaseWalls* has been implemented and exposed via a RESTful interface, together with an event-notification system. Event-notifications can either be *PuSHed* or *Polled*, and effectively formulate the *Case-based Activity Walls*. Process participants can then take actions to interact/manipulate tasks. While at the time of writing the interface provided is programmatic only, future plans are to implement a GUI of the case-walls; or AS IS, we expect the platform to be extensible enabling 3rd-party higher-level and customizable applications.

4.2 Knowledge Graph

The knowledge graph plays a crucial part in the system. It needs to be robust and has to support complex graph query. We use *Neo4j* as the backend DB and

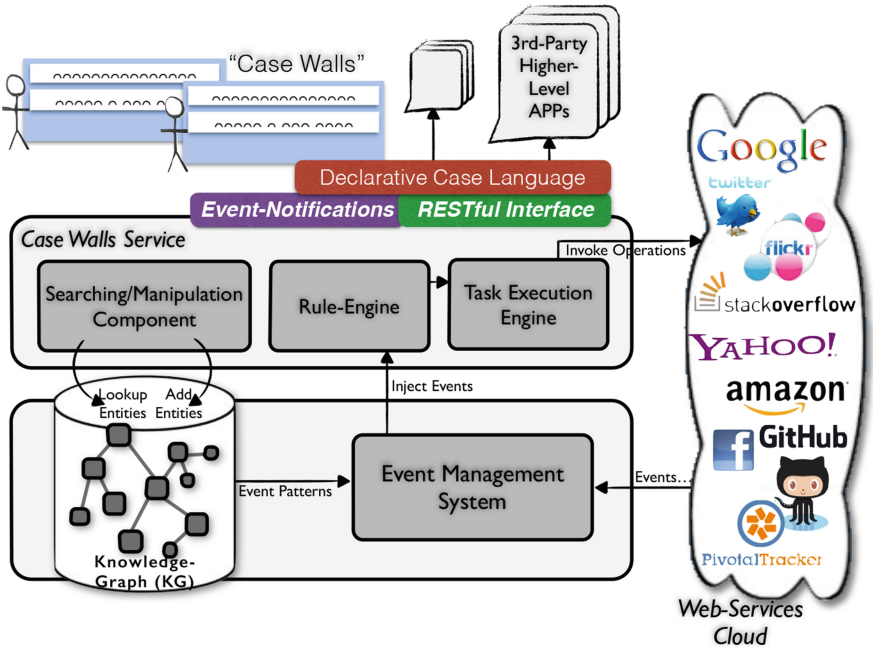

Fig. 7. System architecture of CaseWalls

build a typed graph database with a REST interface on top of it. Neo4j comes with the label system for its entities, which lets the user marks entities with a label but it doesn't enforce any schema except for unique constraint. Therefore we utilized JSON-Schema in the Knowledge graph for validating the JSON data.

By using Neo4j we can use its powerful graph query language call Cypher Query Language to query complex relationships. We can easily find a path from one Task to another. With this capability, we can provide the information about the interoperability between Tasks. We implement a RESTful API to enrich KG itself. Further details of the API can be found via swagger docs[4]. Figure 8 below shows the definition of the `DeliverOnPR` task curated in the knowledge-base.

4.3 Event Management System

The Event Management system is implemented to aggregate and process the events from different services. We use Fluentd[5] for aggregating and dispatching the events received from various services, Norikra[6] and Esper EPL[7] for processing and generating high level events. For instance, defining an event with only

[4] https://raw.githubusercontent.com/freehaha/case-wall-api/master/case.yaml.
[5] http://www.fluentd.org/.
[6] http://norikra.github.io.
[7] http://esper.codehaus.org.

```
Task DeliverOnPR:
{
  "id": "ad6c7cf6-d18b-4323-8bc8-9e56055c313a",
  "name": "DeliverOnPR",
  "description": "deliver a story when a pull request is received",
  "mapping": "{\"current_state\": \"#delivered\",
     \"storyId\": \"$.storyId\", \"projectId\": \"$.projectId\"}",
  "type": "auto",
  "intent": [
    "pullrequest",
    "story",
    "deliver"
  ],
  "_type": "Task",
  "event": [
    "pull_request"
  ],
  "tasks": [
    "UpdateStory"
  ],
  "_created": 1432520794
}
```

Fig. 8. Task definition expressed in JSON for *DeliverOnPR*

the attributes we need (thus masking the different payload and/or structure of the events). Higher-level events also enable defining an event based on a series of targeted events rather than just a single event. Likewise, we utilize MongoDB[8] and ElasticSearch[9] for archiving and indexing event data. For event collection, we implement an event-collecting application that connect to different services. Collected events are then sent to Fluentd to dispatch and process. Since the majority of services today utilize OAuth authorization, we have implemented and allow user to authorize our application to collect events automatically for them. Two kinds of events collecting mechanisms are implemented:

1. **Pushing.** Services like SendGrid, Twilio, Google Drive, Pivotal Tracker let user register a callback url, often called webhook, where the service will send a request to when events arrive. There is also a protocol call *PupSubHubBub* proposed by Google trying to standardize this event delivery method.
2. **Polling.** Some services use the legacy event polling model, requiring the client to constantly check whether there are new events. Another variation of it developed lately is long polling, sometimes called comet, designed to reduce the connection overhead by establish a keep-alive HTTP request. Services providing this kind of mechanisms include Twitter, Twich.tv and Plurk.

4.4 Orchestration Engine: Generating Rules

CaseWalls further layers a more higher-level and declarative case manipulation language (refer Sect. 3). To implement this language, the semantics are

[8] http://www.mongodb.org.
[9] http://www.elasticsearch.org.

translated into Rule-based expressions, denoted: $R_{type} : (Events \rightarrow Actions)$. This effectively means, rules are generated from case definitions. Once deployed, the event-bus can detect relevant event-patterns, and working in conjunction with the rules-engine provision the execution of cases. Since we mentioned there are two main types of tasks: *manual* and *automated*. There are also two corresponding types of rules. Automated-task rules, denoted $R_{automated}$ consist of service-related events (e.g. `PullRequest`); and task-actions (e.g. `DeliverOnPR`). Manual-task rules, denoted R_{manual} may additionally consist of special internal events and actions. While not always necessarily utilized, they are at the disposal of the developer in order to grasp better control over manual tasks. In particular, they may prove useful to manage UI components[10] associated with manual tasks. For example, if an event e_i triggers some manual task to be performed, the rule $Rule_{manual} : (e_i \rightarrow a_x)$ may be defined, where a_x can prepare or perform some pre-processing to some UI component. Likewise, another rule $Rule_{manual} : (e_i.e_j \rightarrow a_y)$ could denote that the manual task has been completed, where e_j is some UI event that the task was completed, and a_y could then be some post-processing action.

To better demonstrate case orchestration rules in the case of automated-tasks, we illustrate as shown in Fig. 9 the series of rules (i.e. `DeliverOnPR`, and `CreateReviewOnPR`) that would generated using the example *Code Review* case that we defined earlier (refer Sect. 3).

```
"rules": [
    {
      "action": [
        "UpdateStory"
      ],
      "event": "pull_request",
      "map": {
        "projectId": "$.projectId",
        "storyId": "$.storyId",
        "current_state": "#delivered"
      }
    },
                                        {
                                          "action": [
                                            "CreateStory"
                                          ],
                                          "event": "pull_request",
                                          "map": {
                                            "projectId": "$.projectId",
                                            "name": "#review PR #{number}"
                                          }
                                        }
                                    ]
```

Fig. 9. Example of generated case orchestration rules

5 Evaluation

Evaluation Objectives. To evaluate overall effectiveness, we assessed the following hypotheses: *CaseWalls* is capable of (a) Improving the ***productivity*** to *model*, *reuse* and *execute* customized service-oriented processes; and (b) Increasing the ***efficiency*** of *application maintainability* for agile service integration.

[10] Even with some 3rd-party tools, developers may tap into the tool (e.g. via Webhooks as in the case of Github). Alternatively, we may also refer here to custom UI components built by the developer to reflect certain manual tasks.

Experimental Setup. To assess the validity, the experiment was conducted by implementing a real-life ***use-case scenario***. Analysis was then conducted via ***comparison*** to other approaches (incl. Javascript, Java, BPEL, Yahoo! Pipes). We divided our scenario into 2 phases; where the latter phase was to add onto the former, thus assessing ease of *maintainability*. Overall productivity was then measured as: (a) Time taken to complete task; (b) Total number of lines-of-code (LOC) excluding white-space; and (c) Number of extra dependencies needed.

Use-Case Scenario *(Code Review and Development Cycle)*. Version Control Systems (VCS) are very common in software engineering - they help avoid collision and improve traceability. While it is important to find where the bug is introduced and revert it, peer review also helps to bring forward discovery of such bugs. *Github* is one of the most popular online open-source repositories for code. Likewise, *Pivotal Tracker (PT)* offers a good story-tracking system, to help the team keep track of their progress. ***Phase 1*** of this scenario involves integration of these two tools in the basic workflow described below:

1. Project Manager PM *creates* a Story and assigned to Engineer.
2. Engineer *starts* working on the Story.
3. Engineer completes programming task and pushes onto Github.
4. Engineer *finishes* and *delivers* the Story.
5. PM *accepts/rejects* the delivery.

Effectively, Github + PT integration may be implemented by parsing commit messages for syntax in the form of: "`#(number)`", such as: *[Starts #12345, #23456] ... [Finishes #12345] ... [Delivers #12345]*. If any such messages are detected, the corresponding action will be performed in PT. For example, if the engineer commit message containing *[Finishes #12345]*, when Github receives this commit, it will automatically *finish* that story in PT. This helps simplify the workflow by eliminating the otherwise manual work done within PT.

While this basic integration provides an initial improvement to eliminate the manual *creation*, *start*, *finish* and *delivery* of a PT "story", ***Phase 2*** involves adapting it to *"continuous integration (CI)"*. The notion of CI, as prominent in software engineering today, calls for "continuous" testing whenever new changes are made. This would thus significantly alter the semantics of the *deliver* action. This means, at *Step 4*, we may want to introduce additional (and iterative) stories (cf. Steps 2–4) for: *testing* and *deployment* before closing this change.

Experimental Results. We set out to prove (or disprove) our hypotheses; the results are illustrated in Fig. 10. Outrightly, BPEL was excluded as real-time events are not available without writing custom extension to the engine. The same applied for Yahoo! Pipes, as it could not receive realtime events via webhooks[11]. Hypothesis H(a) was evaluated as the time to complete both tasks (excluding setup time) and LOC. Using *CaseWalls* this resulted in only 30 mins,

[11] A possible solution could be done using feeds (rss/atom) for receiving the events, and a web-query framework, such as YQL to make requests. However, doing so is less interactive, less efficient and also requires writing sufficiently complex javascript.

compared to an average of 245 mins using other approaches (decrease of $\sim 88\%$); while LOC was 53 compared, to an average of 376 (decrease of $\sim 86\,\%$), respectively. For *CaseWalls* setup included time to create all the Operations/Events/-Tasks in the KG; whereas for Javascript/Java this meant downloading and installing the requisite SDKs (where applicable). *CaseWalls* also resulted in less overall setup time. H(b) was then measured as the cost of implementing *Phase 2* (including any setup time). *CaseWalls* resulted in only 25 mins to implement compared to an average of 220 mins (decrease of $\sim 89\,\%$), with 30 and 93 LOC respectively (decrease of $\sim 67\,\%$).

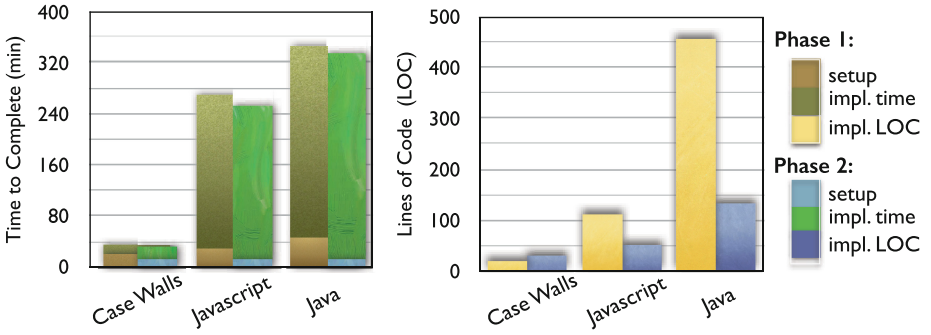

Fig. 10. Evaluation results for GDrive contribution calculator use-case

Overall, it was clear both *time* and *LOC* is significantly reduced when using *CaseWalls*. We thus validate both our hypotheses as true with very promising results. Moreover, our approach did not require any additional libraries, whereas others required on average at least 2–3. *CaseWalls* also provided the facility of increased transparency, as well as agile participant control - compared to other solutions which were rather rigid. In light of these results, this evaluation study successfully demonstrates the anticipated benefit of our proposed approach.

6 Related Work and Concluding Remarks

The ubiquitous access to thousands of APIs offer tremendous potential in modern App development. For example, ProgrammableWeb records some 13,495 APIs over numerous categories, including: financial, mapping, social-networking, etc. The inevitable key to success is thus 'API integration' - as the empowerment to compose disparate services (rather than reimplementing) will reap great reward. Currently, there are a plethora of SaaS-enabled tools that aim to fulfill a specific user-need, however these ready-made solutions often imply conforming to a fixed set of embedded features with little room for user-customization. On the other side of the spectrum, BPM sought to offer customizable process-support over disparate services, albeit it suffered significantly from a lack of flexibility.

In the following, we analyze these two technological polarities. We then offer an innovative set of guiding principles for converging these polar extremes - which is the refreshing outlook we have adopted in positioning our work in this paper.

Web-Services/API Integration Development. Modern service-oriented systems aim to support services integration, [3,8,12]. For instance, the Enterprise Service-Bus (ESB) was an early and still prevalent method for handling message-exchange over heterogenous and distributed components. *Apache ServiceMix* [14], is one example providing advanced features. Micro-services are yet another alternative approach that are well aligned with cloud provisioned services, and tools such as *Netflix Asagard* [15], and *Pivotal Cloud Foundry* [13], have emerged. Albeit, these methods still do not alleviate even professional programmers from being coerced in understanding the various low-level service APIs, as well as working directly with procedural programming constructs to create and maintain complex applications. This leads to an inflexible and costly environment which adds considerable complexity, demands extensive programming effort, multiple and continuous patches, and perpetual solutions, [12].

Process-Oriented Service Programming. Advanced process support systems (e.g. BPEL) and Mashup environments (e.g. Yahoo! Pipes) aimed to counteract the above challenges, while also appealing to the less-technical. However, while they helped avoid low-level API programming, composition environments significantly lacked the productivity support tools that developers were used to whilst programmers using IDEs, (e.g. code search and discovery, ease of reuse, debugging, code generation), [12]. Moreover, they suffered from the lack of flexibility and cannot support run-time changes, [6]. This worsens as the variety of services and variations of application requirements and constraints increase, [4].

Case Management. Flexibility is imperative to transition composition systems from the realm of static and small-scale environments to that of large-scale computing, relying on highly unpredictable and evolving environments. Case-Management is an emerging step in the right directions, given many processes are knowledge-intensive and thereby human-driven, [1,2,9]. For example, a customer initiating a request for some services, the set of interactions among people, e.g. customer and relevant participants, and artifacts from initiation to completion is known as the 'case'. However, while well conceptualized, much of its actual implementation remains vague and depends on its context of use, [7].

Summary. Reflecting on the above, we have discovered Web-services mirroring (at least conceptually), the evolution of database management systems (DBMS)s over the last 30-years. In effect, DBMSs have called for generic abstractions and declarative techniques (e.g., data-models, relational algebras, declarative query techniques) for simplifying the design of complex applications and enabling high level manipulation of data. Similarly, we propose the following set of guiding principles that Web-service APIs should adopt: (i) Modularity, similar building blocks in terms of simple and useful models; (ii) Declarative analysis, including support for high-level language manipulation, integration and transformation; and (iii) Knowledge-preserving, such that API-related programming knowledge can be curated for future communal reuse.

Accordingly, *CaseWalls* is a refreshing step towards this innovative direction, providing a framework to simplify the integration of disparate services and effectively build flexible customizable processes. Our knowledge-driven approach builds upon our previous work [6], and is inspired by efforts in general knowledge-graphs such as "linked data". We have thus proposed a novel *Case Knowledge Graph (CKG)* to facilitate the organization, integration, querying, and reusing of the case management knowledge. Moreover, the ability for case-works to "re-use" process knowledge is a vibrant change to most existing process platforms.

Empowered by this knowledge graph, we also provided a novel case customization and deployment platform. And to even further increase user efficiency, we introduced a simple, declarative yet powerful language to query and analyze the knowledge graph. Unlike any previous works, *CaseWalls* focusses on the transparency aspect of mid-process knowledge. The concept of "walls" thus act as an activity wall akin to social status updates, albeit instead updates are sourced as relevant events from case tasks. Participants are then empowered to track the case execution, and react/interact accordingly.

Experimental results shows promising results, in particular addressing the dimensions of increased user-efficiency. In future, we are excited to enhance and extend the language power and expressivity - as well as implement a novel graphical user-interface that mimics social-networking platforms. Whereby case-based process functionality can effectively be combined within everyday tasks. We are therefore very optimistic this work provides the foundation for future growth into a new breed of enhanced process-support.

References

1. Van der Aalst, W.M., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data Knowl. Eng. **53**(2), 129–162 (2005)
2. Swenson, K., et al.: Taming the Unpredictable Real World Adaptive Case Management: Case Studies and Practical Guidance. Future Strategies Inc. (2011)
3. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts Architectures and Applications. Springer Publishing Company Incorporated, Heidelberg (2010)
4. Barukh, M.C., Benatallah, B.: ServiceBase: a programming knowledge-base for service oriented development. In: Feng, L., Bressan, S., Winiwarter, W., Song, W., Meng, W. (eds.) DASFAA 2013, Part II. LNCS, vol. 7826, pp. 123–138. Springer, Heidelberg (2013)
5. Barukh, M.C., Benatallah, B.: A toolkit for simplified web-services programming. In: Lin, X., Manolopoulos, Y., Srivastava, D., Huang, G. (eds.) WISE 2013, Part II. LNCS, vol. 8181, pp. 515–518. Springer, Heidelberg (2013)
6. Barukh, M.C., Benatallah, B.: *ProcessBase*: a hybrid process management platform. In: Franch, X., Bhiri, S., Ghose, A.K., Lewis, G.A. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 16–31. Springer, Heidelberg (2014)
7. Böhringer, M.: Emergent case management for ad-hoc processes: a solution based on microblogging and activity streams. In: Muehlen, M., Su, J. (eds.) BPM 2010 Workshops. LNBIP, vol. 66, pp. 384–395. Springer, Heidelberg (2011)

8. Geambasu, R., Cheung, C., Moshchuk, A., Gribble, S.D., Levy, H.M.: Organizing and sharing distributed personal web-service data. In: Proceedings of the 17th International Conference on World Wide Web, pp. 755–764. ACM (2008)
9. Kaan, K., Reijers, H.A., van der Molen, P.: Introducing case management: opening workflow management's black box. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 358–367. Springer, Heidelberg (2006)
10. Klemisch, K., Weber, I., Benatallah, B.: Context-aware UI component reuse. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 68–83. Springer, Heidelberg (2013)
11. Nolan, D., Lang, D.T.: Authentication for web services via OAuth. In: Nolan, D., Lang, D.T. (eds.) XML and Web Technologies for Data Sciences with R, pp. 441–461. Springer, New York (2014)
12. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. big web services: making the right architectural decision. In: Proceedings of the 17th International Conference on World Wide Web, pp. 805–814. ACM (2008)
13. Pivotal-Cloud-Foundry. http://pivotal.io/platform-as-a-service/pivotal-cloud-foundry
14. ServiceMix, A.: Apache servicemix 3. x users' guide. Apache ServiceMix Community (2007). http://incubator.apache.org/servicemix/users-guide.html. (Cited on p. 72, 73 and 149)
15. Sondow, J.: Asagard: web-based cloud management and deployment. The Netflix Tech Blog (2012)