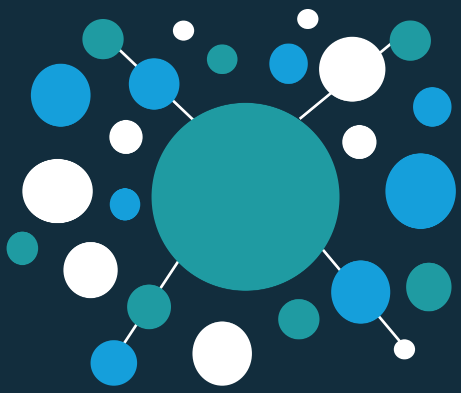
SPARKL®

Microservices Orchestration with the SPARKL® Sequencing Engine



Introduction to Microservices

Microservices are lightweight, semi-autonomous, self-sustaining pieces of logic which operate in parallel and collaborate through network-based message passing. The concept is not a new one, but it's an approach that's being used more and more in enterprise architecture.

For example, challenger bank Monzo have announced they had made the unusual decision to build their backend as a collection of distributed microservices. Larger enterprises normally build their applications on a single technology platform, called monoliths. These are easy to build, but harder to manage over time.

At SPARKL, we call microservices "black boxes". They do something, but it doesn't really matter how. The difficulty comes when you combine them. It's important to understand the various behaviours of the pieces of your system as it gets bigger. That's a typical problem for any bank, which can have thousands of applications running over tens of thousands of systems.

By splitting a centralised system into microservices, companies like Monzo have a choice when it comes to building a service on their applications. Developers are able to work on different areas of an application without interrupting whatever is going on elsewhere. This sets up an ongoing, day-to-day relationship with the technology and its users.

The main principles of microservices are:

- **Loose-coupling** - there should be limited dependencies between microservices, so changes made to one service shouldn't have an impact on other services, nor should there be chatty communication between services, though they should be asynchronous.

- **Smart end-points** - the "***keeping the smarts*** (i.e. logic) ***in the end-point whilst keeping the middleware dumb***" principle of microservices. Services should be built on minimal assumptions and constraints concerning the nature of the environment in which they operate.

- **High cohesion** - microservices should be as small as possible while still maintaining high cohesion. Related logic should be placed within the same microservice, whilst unrelated logic should not be. The cost to replace small services with a better implementation, or delete them altogether, is much easier to manage.

With microservices, every business function can be customised - the system can implement different **quality-of-service** practices for the various functions.

Many **downstream** services can be used to satisfy a business transaction, which starts with a call on a frontline service. For example, many checks and other actions could be taken when a customer registers for an online service.

When an error occurs in the registration process, the ops team drills down and looks at individual calls to downstream services, leading to the error. The team **correlates** log entries corresponding to the same transaction, finding the specific call to a downstream service that caused the transaction to fail.

Being able to correlate events that take place in the satisfaction of a transaction can be particularly challenging with microservices, as the logic is dispersed potentially between several services.

In the following brief, we will indicate how a transaction-based audit trail, showing correlations between events, is made available through **process-based data provenance.**

SPARKL for Microservices Orchestration

SPARKL is powerful technology for managing the behaviour of distributed systems. It's primarily used for orchestration, a.k.a sequencing. It records the operations as data events, and sequences them together to satisfy a transaction.

A user can configure SPARKL in **application** terms through services, along with various operations that can be performed on those services. Whenever a **transaction** is initiated, SPARKL plans its satisfaction using the specified application configurations. It routes messages between microservices according to these plans.

Service orchestration in SPARKL focuses primarily on supporting application flows. This means that, in the limit, service infrastructure may be provisioned and torn-down per individual call made against SPARKL, as depicted in **Figure 1**. In reality, there will be a balance between per-operation service artefacts, those that last for the lifetime of the overall service deployment, and others in between.

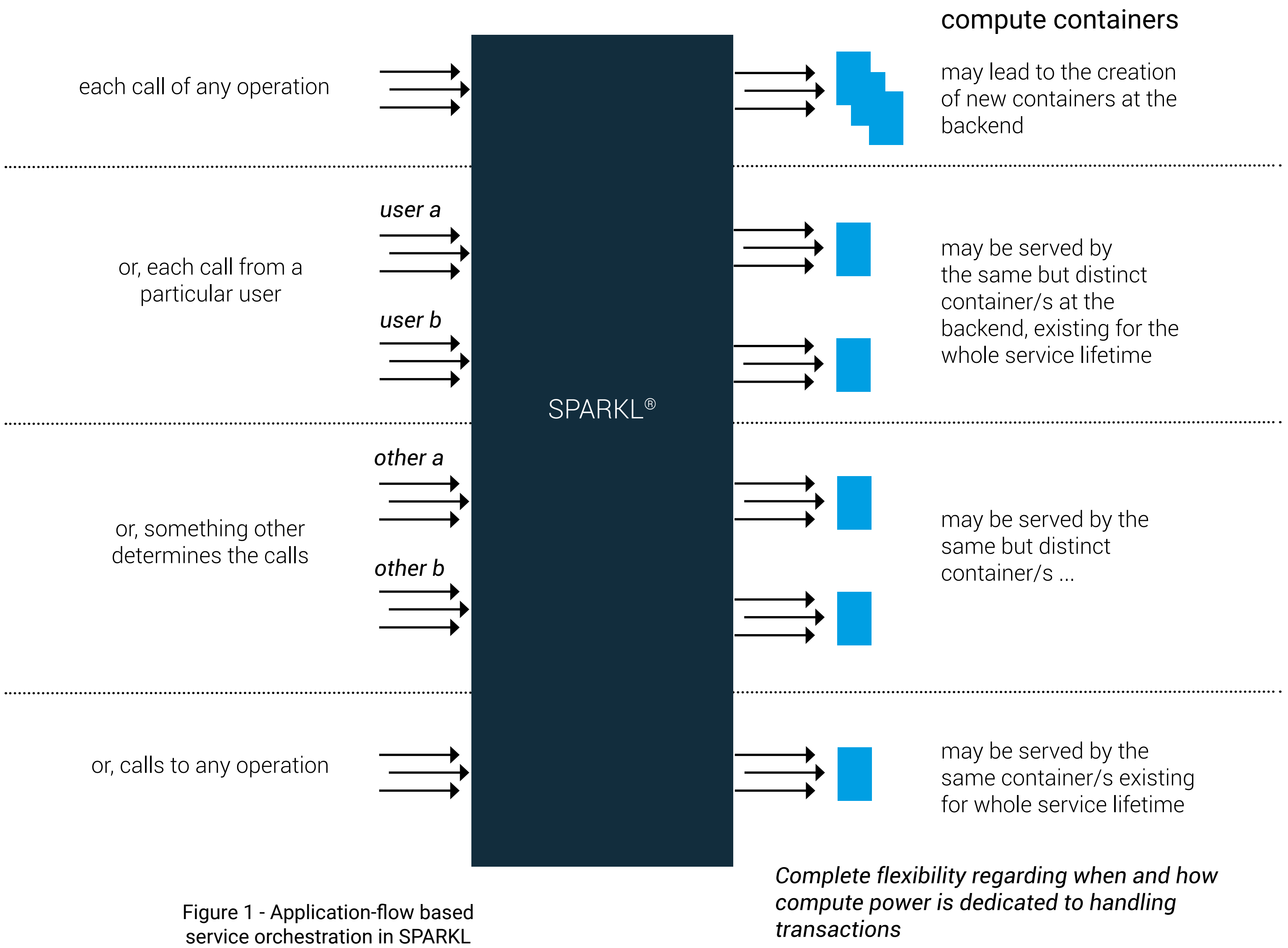

compute containers
each call of any operation
may lead to the creation of new containers at the backend
user a
or, each call from a particular user
user b
may be served by the same but distinct container/s at the backend, existing for the whole service lifetime
SPARKL®
other a
or, something other determines the calls
other b
may be served by the same but distinct container/s ...
or, calls to any operation
may be served by the same container/s existing for whole service lifetime
Complete flexibility regarding when and how compute power is dedicated to handling transactions

Figure 1 - Application-flow based service orchestration in SPARKL

Provisioning per application-flow works for:

- Individual transactions that are long-running (reducing infrastructure set-up latency and costs)
- Per-user or some other determinant, in order to fine-tune performance
- Contexts that are sensitive in terms of its security requirements, for which **clean** infrastructure would be worthwhile.

Application-flow provisioning works particularly well in the microservices model because of the fine-grained nature of the services themselves. SPARKL can reason if a microservice is needed at all - e.g in the satisfaction of a transaction; whether the number of instances of a microservice should be scaled out because of long transaction times; or to implement different qualities of service for different users at different times, for example.

A transaction can be executed on a system, using SPARKL, according to these two patterns:

- The `solicit->response` pattern describes the **solicitation** of a transaction to achieve a desired state, reflected in the response.
- The `notify->consume` pattern describes the handling of an event, of which SPARKL has been **notified**. SPARKL eagerly looks to inform as many **consumers** as it can of the event occurrence, even performing intermediate operations in order to reshape the original event prior to consumption.

Essentially, the first is a synchronous pattern, and the second, an asynchronous pattern.

Figure 2 shows a possible instantiation of the `solicit->response` pattern.

SPARKL will look to satisfy the transaction, embodied as a **solicit data** event sent to SPARKL, by sequencing (request) operations on various service end-points.

The solicit has a single possible response, other than error, which is a value for the Red data field. The solicit is supplied with Blue and Green data field values.

SPARKL sees that, in order to satisfy the solicit, it needs to sequence two operations. This is so that a value for the Yellow data field can be obtained given a Green data field value.

Then, Blue and Yellow field values are given to the `request_balance` operation, offered by a back-end service. This service gives back a value for the Red data field, which is routed back to the original caller, as a response data event.

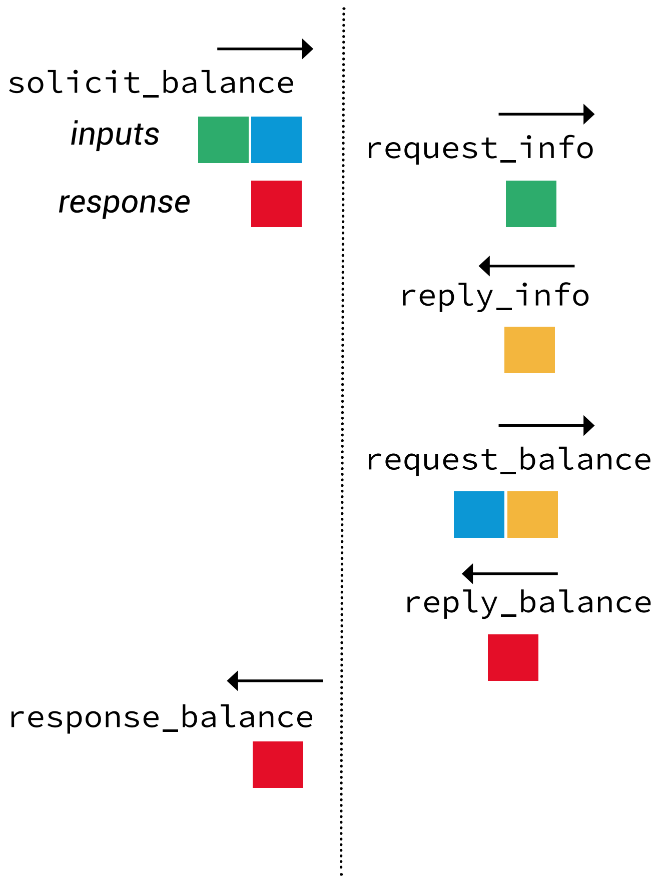


Figure 2 - SPARKL sequencing the satisfaction of a solicit transaction

In summary, SPARKL plans out the execution of operations to satisfy a solicit event, such that the operations yield sufficient data values for one of its responses.

The orchestration taking place is captured as a mix, which is a small, self-contained definition of **solicit** and **request operations** on **services** and **data fields**.

The satisfaction of the solicit transaction entails six data events, which are logged for auditing purposes by SPARKL:

- the initial solicit, and its response
- two sets of request and reply events.

Figure 3 shows a possible instantiation of the `notify->consume` pattern. When an event notification is received by SPARKL, the service contributing `consume_blue` (in the implied mix) can immediately be called. This is because the Blue data field is immediately available, and is sent in a **consume** data event to the service.

SPARKL works out that the particular service contributing the `consume_yellow` operation can also be called if SPARKL first inserts the `reshape_green` operation in order to elicit the Yellow data field.

The purpose of **request** operations, in the `notify->consume` pattern, is to reshape **notified** events prior to **consumption** by service end-points.

The `notify->consume` pattern leaches into the `solicit->response` pattern, but not vice-versa.

Consume operations may be invoked in the course of satisfying a solicit transaction to provide field data from the ongoing transaction.

This data is, for informative purposes, to be consumed by the service end-point.

Thus, there is a `solicit->consume` aspect to the `solicit->response` pattern.

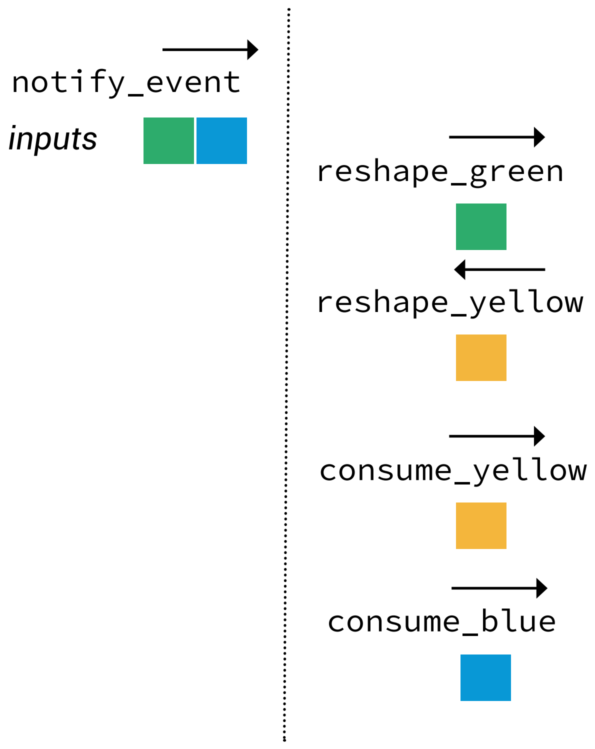


Figure 3 - SPARKL sequencing the satisfaction of a notify transaction

An ideal application of SPARKL is in the orchestration of microservices. Communication between services happens mainly according to the `notify->consume` pattern of SPARKL, promoting **loose-coupling**, although the initiating call may be a `solicit->response`.

SPARKL further encourages loose-coupling as its ***fieldset-based semantics***, which naturally allows for changing service APIs. For instance, an operation originally may reply with Red and Blue data fields. If a consumer requires just Red, then it still gets called by SPARKL if the operation changes to returning Red and Green fields, as it just needs Red.

SPARKL is able to push control logic into the services themselves, as part of the **put-the-smarts-in-the-end-points** philosophy of microservices.

This helps with resiliency and loose-coupling. An instance of SPARKL could potentially be placed in every microservice, as depicted in **Figure 4**, to orchestrate operations and communications between processes running on the microservice, and other microservices.

Figure 4 - SPARKL-based microservices

Given that SPARKL is baked into the microservices themselves, there is natural redundancy and resiliency in the approach - if one SPARKL is not currently available to do **load balancing**, or instruct **autoscaling**, say, then another steps in. There is no need for a centralised controller.

Developers of individual microservices produce individual interface definitions as SPARKL mixes. These mixes can be arbitrarily composed to describe the behaviour of an overall system, leaving glue logic out of the process - a simple cut-and-paste of mix definitions to create compositions **just** works. Again, this means that the smarts are owned by the services themselves.

In a legacy setting where only **some** of the system components are implemented using microservices, a sprinkling of SPARKL is enough to immediately see benefits with respect to capturing data provenance, from which insights can be drawn through analytics. We offer a solution where data provenance can be captured against both non-SPARKL and SPARKL-based service components.

Data provenance is concerned with tracking the flow of data through service end-points including how it is shaped. We often want to assess this at a transactional granularity, hence, we use the term **transaction-oriented data provenance**.

Banks are obliged to report on every aspect of their operations, yet today's technology's stack simply doesn't provide the detailed, comprehensive log data required to satisfy the demands of regulators and shareholders.

There are two types of **data provenance** that SPARKL implements for microservices, both highly useful for this kind of problem. **Process Provenance** captures the history of operations carried out on system components as well as the resulting manipulation of data records stored on systems. It captures specific events of reading and writing of such data.

With Query Provenance, the satisfaction of a query made on data stored by a system requires subqueries to be carried out across many disparate data sources, involving one or more ETL layers, different database views or rules engines.

- Query provenance is the descriptions (in terms of metadata) of the data records used to satisfy a query, and a graph of how the data records are combined for this purpose.
- Queries are satisfied rather than an account of specific queries made against a system, which would fall under process provenance.
- The metadata of any data record may include the process provenance which details how the data record has thus far been shaped by the back-end systems.

Any system architecture - whether it's using microservices or not - needs to support reporting along these lines.

There are the following possibilities when deploying SPARKL in a microservices context, as depicted in **Figure 5**, where instances of either may exist for any particular system.

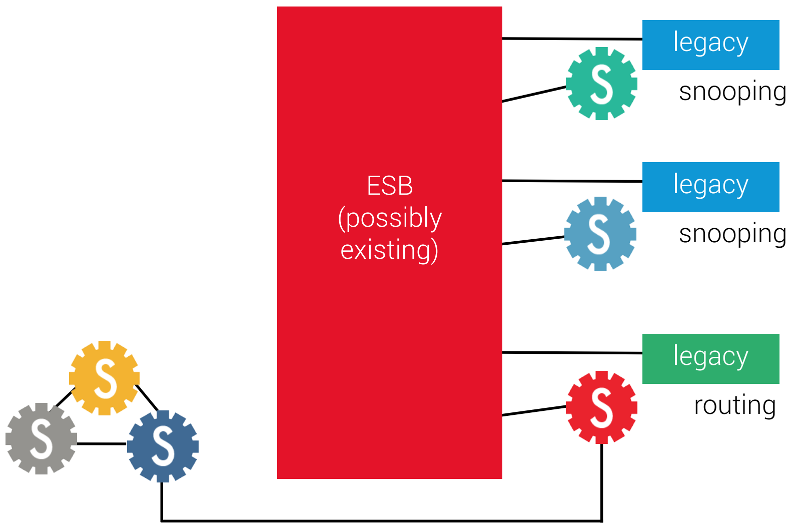


Figure 5 - Supporting legacy systems components and SPARKL microservices

SPARKL-based microservices can communicate with other microservices via standard SPARKL sequencing and Erlang RPC mechanisms, moving from one SPARKL node to another.

SPARKL can be baked into a microservice to directly orchestrate operations on the microservice, as well as forward solicitations or notifications from the microservice. It logs all these operations with an audit trail.

In some cases, the system might use a microservice or a legacy system component without SPARKL. Here, a shadow SPARKL microservice would be deployed for the system in order to extract information for provenance tracking.

We make use of a message bus solution to act as a message protocol interchange mechanism. The legacy system component is coerced to communicate directly with the bus. The choice of bus is made based on the system components it needs to support in terms of the protocols they use. E.g. RESTified HTTP, other HTTP, or WSDL/SOAP.

Two further options are possible.

- The shadow microservice is limited to snooping on traffic routed by the message bus. The microservice will oversee (possibly offline) processing of these messages, extracting information for process provenance. This can occur when two non-SPARKL components are communicating directly through the message bus.

- Or the SPARKL-based shadow microservice serves wire the non-SPARKL system component up with other SPARKL-based microservices. In this case, the communication gets switched into the SPARKL network, shown on the left of **Figure 5**. The mix enabling this configuration will capture how the traffic to and from the component should be marshalled into SPARKL operations. Once the traffic is captured as SPARKL operations, full SPARKL-based logging follows.

So even for legacy system components, which use a range of protocols, we immediately get value in terms of process-based data provenance by **sprinkling a little SPARKL**.

Every microservice provides a specification of health checks (as another mix) which SPARKL routinely applies to ensure the health of the service. In the event that a microservice instance is misbehaving according to the execution of a health check mix, SPARKL will replace it.

In summary, SPARKL is a complete legacy which can be employed immediately into enterprises and start providing value, with minimal overhead. It's a complete legacy system, where process

- SPARKL supports fine-grained (per-user, per-some other criterion) provisioning, load balancing and auto-scaling of microservices. It is capable of enforcing SLAs because of this.
- SPARKL supports Devops and continuous delivery patterns typical of a microservices orchestrator, including **circuit breakers**, **green/blue deployments**, **A/B testing** and **canary releases**.
- SPARKL is scalable software, implementing a **distributed intelligence** pattern where a SPARKL router is placed into every microservice.
- SPARKL natively supports Erlang, Python and script and browser-based nanoservices.

Bonus - SPARKL doesn't prescribe technology stacks, but it **is** technology-agnostic. In **Figure 6**, we show one possible technology integration stack with SPARKL. One is a business process **execution** stack, and the other is an analytics stack.

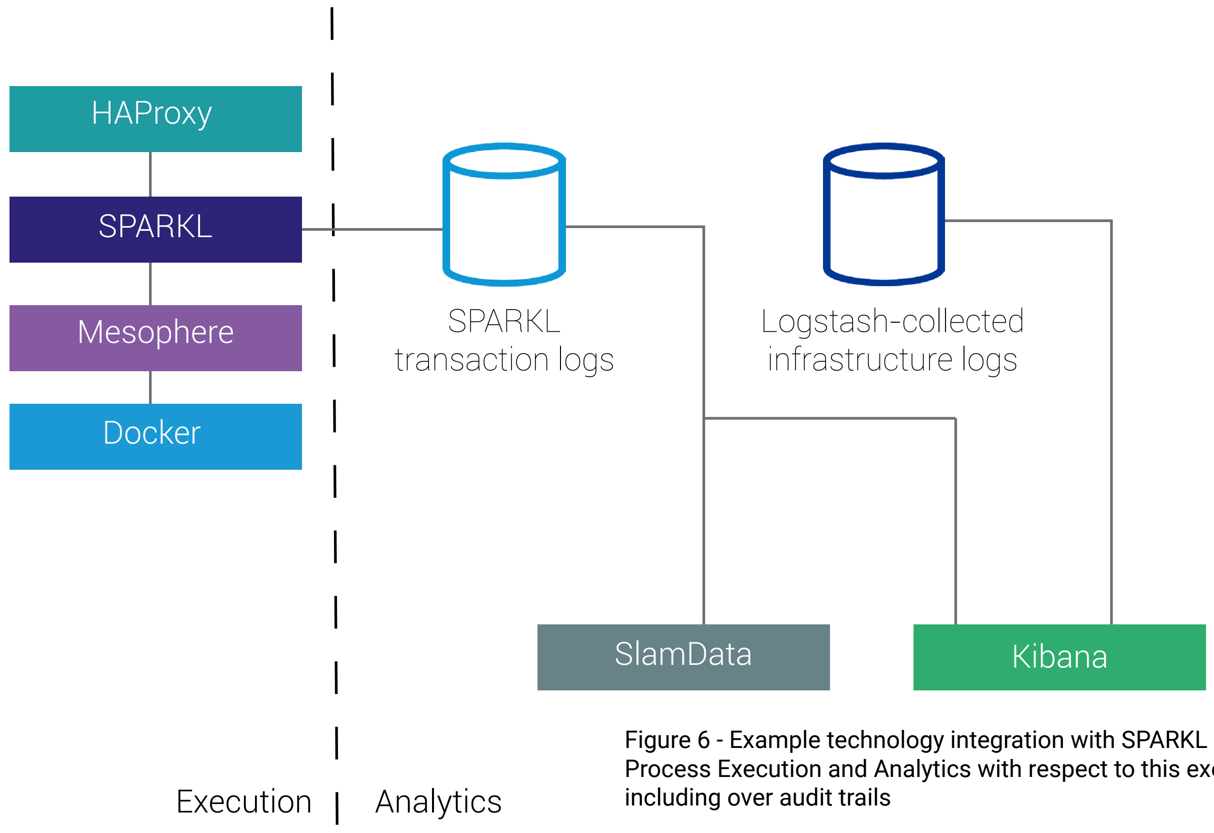


Figure 6 - Example technology integration with SPARKL for both Process Execution and Analytics with respect to this execution, including over audit trails

Let’s talk

- Need more info? Drop us an e-mail at talk@sparkl.com
- See SPARKL tutorials and demos at sparkl.com/docs