# Java Development Task: Dashboard for Github activity data

**Key business driver**: Enable business stake holders to understand the value of their developers’ output.

Suggested KPIs:

1. Viewing Github commits per developer over a specified date range
   1. Total up number of commits
   2. Total lines of code
   3. Total of comments added (indicator of adding value to the team)
   4. Idea: Release metric - what % of committed lines of code do not exist in the released product (indicator of wasted time)
   5. Number of pull requests raised
2. Trending of activity listed above: are we seeing changes in behavior that might indicate a change in engagement in the developers.
3. Team / Function aggregation – grouping the above information into entire Teams for team comparison. Subgroup by function (eg dev and test)
4. Comparison of activity on different repos against each other – are certain projects being worked on more actively than others.

Architectural Approach Options

(assuming that, in a real-world situation, GitHub “Insights” functionality is not sufficient):

1. Direct Github > Elastic connector (eg [this one)](https://github.com/elastic/connectors/blob/8.15/connectors/sources/github.py) with a Kibana frontend to visualize the data.

Very much a low-code solution and avoids creating new self-managed product estate.

* 1. **Investigation** into relative costs of cloud hosted Elasticsearch, Kibana, and any restrictions on the connector linked to above (can any such restriction be overcome with a [custom connector](https://www.elastic.co/guide/en/elasticsearch/reference/8.15/connector-apis.html)?)

Visualization options are highly configurable in [Kibana,](https://www.elastic.co/guide/en/kibana/8.15/introduction.html) and this solution would remove most of the actual development costs of providing the solution.

**Subject to a cost-vs-reward analysis against option 2, this option appears to be the strongest** considering:

1. Speed of delivery
2. Initial developmental cost outlay
3. Self-managed estate overhead (security / monitoring / DR etc)
4. Bespoke coded solution involving Java-based Spring-Boot microservices and a frontend.

If Option 1 is considered undesirable (potentially due to ongoing cost reason, or wanting greater self-management of the product’s stack) then the following architectural points will need to be considered:

* 1. Github client
     1. The GitHub documentation pages [here](https://docs.github.com/en/rest/using-the-rest-api/libraries-for-the-rest-api?apiVer) document option for pre-existing Java Clients for confusing GitHubAPI data. Of the two, the [github-api.kohsuke.org](http://github-api.kohsuke.org) option seems to be the only viable option.

For the PoC, add for trialing the capabilities of Kibana dashboards, a simple rest client obtaining the full json response payload from GitHub has been used, but with refined requirements, and investigation into performance of that solution compared to a full GitHub Java Client, this might want to be revised.

* + 1. **Investigation**: What forms of data return are available here? Can we get a dump of large sections of data in one go or is information partitioned to require specific searches?
    2. Data persistence – depends on transaction load and data quantity.
       1. There will be an advantage **caching data** to avoid repeated calls to Github for previously retrieved data
          1. In memory – simple but overlying volatile for production – only useful in PoC
          2. Elasticsearch itself – the store is already there, could create a second index for the caches, seperate from the actual GitHub data.
          3. Separate store technology if ES is unstuiable in someway (consider an AWS DB solution)
    3. Data-on-request vs webhooks/callback
       1. Data-on-request is the **lightest weight** option and may be suitable given low estimated usage of the dashboard (e.g. a low usage internal team tool), or a focus on cost and speed of delivery of the dashboard. In this instance caching is likely to be of use to avoid duplicated requests for the same information.
       2. **Webhooks** seem available to allow callbacks for many (all?) GitHub actions e.g. [push.](https://docs.github.com/en/webhooks/webhook-events-and-payloads#push) This is the preferred solution where we are:
          1. Expecting significant usage of the dashboard system
          2. Requiring fully accurate live reporting of data (seems unlikely given the subject matter?)
          3. Needing to keep transactions to Github to a minimum due to speed-of-response considerations or comms stabilizing issues.
  1. Statistical analysis of Github data will be required. The level of detail here will depend on granularity of information desired. Options will include:
     1. Bespoke java-coded query handler that provides for specific query types e.g “selected stats for GitHub User ‘X’ between dates ‘Y’ and ‘Z’”
     2. **Leveraging Elasticsearch** – data from GitHub being added to an Elastic index either hosted in the cloud or deployed locally (docker is an option here). As above, cloud costs of Elastic hosting need considering against self-managed Elastic deployment. Elasticsearch is not preferred as a main database, so suggest an alternative DB solution as the primary, the searchable parts of which are then synchronized to Elasticsearch.
  2. Visualization frontend will be required. The options here range from zero-code to code-heavy:
     1. Kibana, twinned with an Elasticsearch store, can provide visualization without any true coding overhead. The PoC will use this option for it’s speed to delivery.
     2. React based reactive frontend served from the Spring-boot backend. This will nicely tie the Spring-boot service code into the presentation layer, though it does introduce a team dependency on React/JScript knowledge. This approach benefits from allowing leveraging of [ElasticUI react components](https://eui.elastic.co/#/elastic-charts/creating-charts) provide charting components that can be leveraged in a React frontend. Requires a React app deployment but will provide a high-quality look and feel.
  3. Security. Leverage Spring security implement Oauth2 or login credential security to prevent access to unauthorized users.
  4. Scaling and future growth.
     1. Vertical:
        1. Initial focus would be on caching as discussed above.
        2. Creating a thread-pooled resource of non-blocking jobs to pull/digest data (depending on approach taken)
     2. Horizontal:
        1. scaling can be achieved by replicating the API across multiple instances, such as Kubernetes containers.
        2. Load-balancing can be achieved through orchestration, such as with.