Human Brain Project

Co-funded by the European Union

Project Number:	720270	Project Title:	Human Brain Project SGA1

Document Title:	HBP Software Engineering and Quality Assurance Approach
Document Filename:	D11.3.3 (D62.2 D17 - SGA1 M10) ACCEPTED 180709.docx
Deliverable Number:	SGA1 D11.3.3 (D62.2, D17)
Deliverable Type:	Report
Work Package(s):	WPs 5.5, 5.8, 6.4, 7.1, 7.5, 7.6, 8.6, 9.5, 11.3 (WPs involved in writing this document)
Dissemination Level:	PU (= public)
Planned Delivery Date:	SGA1 M10 / 31 Jan 2017
Actual Delivery Date:	Submitted: 30 Nov 2017 (M20), accepted 9 Jul 2018

Authors:	Alois KNOLL, TUM (P56), Jeff MULLER, EPFL (P1),
Compiling Editors:	Jeff MULLER, EPFL (P1)
Contributors:	Carlos AGUADO, EPFL (P1), Olivier AMBLET, EPFL (P1), Yury BRUKAU, EPFL (P1), Ludovic CLAUDE CHUV (P27) Jean-Denis COURCOL, EPFL (P1), Mihaela DAMIEN CHUV (P27), Andrew DAVISON, CNRS (P10), Michael GEVAERT, EPFL (P1), Valentin HAENAL, EPFL (P1), Thomas HEINIS, ICL (P28), Samuel KERRIEN, EPFL (P1), Anna LÜHRS, JUELICH (P20), Hans PLESSER, NMBU (P44),), Colin McMURTRIE, ETHZ (P18), Ben MORRICE, EPFL (P1), Luis RIQUALME, EPFL (P1), Andrew ROWLEY, UMAN (P63), Martina SCHMALHOLZ, UHEI (P47), Felix SCHUERMANN, EPFL (P1), Stefano ZANINETTA, EPFL (P1),
SciTechCoord Review:	EPFL (P1): Jeff MULLER, Martin TELEFONT UHEI (P47): Martina SCHMALHOLZ, Sabine SCHNEIDER
Editorial Review:	EPFL (P1): Guy WILLIS, Martin O'NEILL

Abstract:	This document summarises the current HBP Software Engineering and Quality Assurance approach as well as the principles upon which the approach is based. It describes in some places the consensus-driven standardised approaches to software development and testing practices which is sensitive to the realities of research software development and the needs of infrastructure construction. Due to the heterogeneous nature of the developments in HBP, there are places where a single standard is not appropriate and this document describes multiple variants. It is not expected that these practices will be enforced globally, but will serve to guide software developers who envision their software should be part of the HBP infrastructure offering.
Keywords:	system engineering, software engineering, quality assurance

Table of Contents

Software Engineering and Quality Assurance 5
1. Introduction 5
1.1 Status 5
2. Principles 5
3. Technical Management 6
4. HBP Agile Practice 7
4.1 Scrum: Roles and Procedures 7
4.2 Kanban 10
4.3 Integration of Larger Projects 12
5. Software Engineering 12
5.1 Software Development 12
5.2 Operations Standards (DevOps) 13
5.3 HBP Standard DevOps Stack 15
5.4 BBP Standard DevOps Stack 16
5.5 Medical Informatics Platform Standard DevOps Stack 17
5.6 Neuromorphic Platform Standard DevOps Stack 18
5.7 HPAC Platform Standard DevOps Stack 21
5.8 Long-term software maintenance 21
6. Quality Assurance 21
6.1 Unit Testing 22
6.2 Integration Testing 22
6.3 Manual Testing 22
6.4 User Interface Testing 23
6.5 Monitoring 23
6.6 Platform-testing specifics 23
7. Recommendations 24
7.1 Communication 24
7.2 Quality assurance 27
8. Systems Documentation 29
9. User Support 29
Annex A - HBP Extended TRL Standards 31
Annex B — Infrastructure Tiers 33
Annex C - BBP Development and Deployment Standards 34
Annex D - BBP Python Development Standards 97
1 Introduction 101
2 The Ipython Interpreter 101
3 Style Guide and Conventions 102
3.1 Style consistency 102
3.1 Code lay-out 103
3.3 Imports 104
3.4 Whitespace in Expressions and Statements 108
3.5 Naming Conventions 110
3.6 Exceptions 113
4 Static Checking 114
4.1 Comparison 115
4.2 Example 115
5 Testing 117
5.1 Unitest 118
5.2 Nosetest 119
5.3 Numpy Testing 119

6 Documentation 120
6.1 Docstrings 120
6.2 Numpy Docstrings 121
6.3 File Header 122
6.4 Automatic API Generation 122
7 Writing the `setup.py` file 122
8 Versioning 124
9 Obtaining Python Packages 125
9.1 Python on Linux 125
9.2 Python on Windows and Mac 126
10 Wrapping Native code 126
11 Performance Considerations 127
11.1 Define "Too Slow" 127
11.2 Identifying the Bottlenecks 127
11.3 Avoiding Premature Optimization 128
11.4 Consider time for Development 128
12 Target Version 128
12.1 Python 2.x 128
12.2 Python 3.x 128
Annex E - UI Development and Testing Standards 133
Overview 133
HBP Collaboratory 133
Step 1 – Concept 133
Step2 – Description of users needs 134
Step3 – User validation 134
Step4 – Mockups 134
Step 5 – User validation 135
Step 6 (part 1) – Enter stories in the backlog 135
Step 6 – Front end prototype 136
Step7, 8 – Testing and Refinement 136
Step 9 - Deployment on production 136
Subsequent steps 137
SP10 Example 137
Step 1 – Concept (driven by User need) 137
Steps 2 and 3 – Description of User need 137
Step 4 - mockup 138
Step 6 – Part 1 – enter story 1 into the backlog 139
Step 6 – Part 1 – enter story 2 into the backlog 139
Step 6 – Part 1 – enter story 3 into the backlog 140
Step 6,7,8 – In sprint usability testing and review 141
Subsequent steps 141
SP8 - Feature development process 142
Annex F - Software Development Committee Charter 143
Charter Working Group 143
Purpose of the Software Development Committee (SDC) 143
Mandate of the SDC 144
Annex G - Infrastructure Development Committee Charter 145
Charter Working Group 145
Purpose of the Infrastructure Development Committee (IDC) 145
Mandate of the IDC 145
Annex H: References 147

List of Figures

Figure 1: Agile Scrum Iteration Workflow: from the Product Owner Definition of the Backlog to a Finished Product 8
Figure 2: Risk Reduction using a Short Feedback Loop with Users 9
Figure 3: Kanban kick start: Estimation 11
Figure 4: Agile iterative integration 12
Figure 5: HBP standard service deployment 14
Figure 6: HBP support channels 30

List of Tables

Table 1: HBP Standard DevOps Stack 15
Table 2: BBP Standard DevOps Stack 16
Table 3: Medical Informatics Platform Standard DevOps Stack 17
Table 4: Job Queue and Neuromorphic Collaboratory Integration 18
Table 5: SpiNNaker Large Machine Service 19
Table 6: NMPM-1 (20-wafer BrainScaleS System) Service 20
Table 7: HBP Support count, per SP 29
Table 8: Technology Readiness Levels 31

Software Engineering and Quality Assurance

1. Introduction

One of the primary goals for the Human Brain Project (HBP) is to build an infrastructure that serves cutting edge neuroscience, within the HBP and beyond. This infrastructure will be made up of software, services and hardware, working in harmony to enable use cases that expand neuroscience and to apply that knowledge to design novel hardware, improve robots for a myriad of applications and to discover new medical treatments. Some of the software and services needed will be custom built for the Project. Some of this infrastructure will be built by integrating and customising existing software and services. As a result, the HBP must undertake a substantial collection of software and system engineering activities, built on best practices from industry and customised to HBP needs, to ensure that infrastructure is developed in a cost-effective and user-centred manner.

This document describes the principles, tools, services, and development practices to be used in the development of the HBP Platforms. It covers both the Agile and more traditional software development practices used throughout the Project, and also the software development infrastructure used by the various Subprojects (SPs) to achieve efficient, robust, and well-tested software and services. Furthermore, this document gives an overview of the testing practices used throughout the Project in all phases of software development and service deployment.

1.1 Status

This document reports status at the time of drafting. It will be updated at least once per Specific Grant Agreement (SGA) and also when development practices inside the project are substantially modified.

2. Principles

The fundamental principle for all software produced for the HBP is that it will be of the highest quality, balanced by the need for timely release of new functionality required by users. This means that, at any given time, there are many software efforts in the Project which are clearly in their early stages and as a result have been released to a limited set of early users on a pre-release basis.

The following principles also apply:

1) Subsidiarity:
   a) Each Platform SP is responsible for the coordination of development, user recruitment and infrastructure management of their Platform.
   b) User recruitment and infrastructure management for the Collaboratory serves as an umbrella to coordinate activities between and across Platforms.
   c) Subsidiarity of the respective Platform SPs is compatible with the current governance and planning models of HBP.
2) Quality of the Platforms:
   a) Given the high expectations and visibility of the HBP, all HBP Platforms must deliver a positive experience in order to encourage early adoption.
   b) Each Platform will first offer services in those areas with the highest potential for their respective community.
   c) The quality of offerings will be tailored to balance the need for early release against the expectations of quality for a particular user community. Many user communities prefer very early release of less mature software to later release with higher quality.

Platform SPs have taken this into consideration in their Platform development roadmaps.

3) Agile best-practices are recommended, but may be implemented selectively:
   a) The Platforms teams are faced with a challenging product development problem, but the difficulties involved can be reduced by regular access to representative customers in the various HBP SPs.
   b) Iterative development, Continuous integration and DEVelopment and OPerations (DevOps) are well suited to the development of robust software and services by small focused teams.
   c) The Project Lifecycle referred to in this document is described both in the Framework Partnership Agreement (FPA) and first Specific Grant Agreement (SGA1). It is a Lean Management approach to scale coordination of projects using Agile methodologies across the HBP.

3. Technical Management

Each Platform SP is responsible for planning the resources for operating and supporting its Platform and its users. This follows from the subsidiarity principle. Resources should be planned according to the estimated demand and expected maturity requirements, in alignment with the scientific needs of the respective domains. These estimates should be produced with help from the Community Coordinator in each SP and on the basis of the Platform roadmap, the current usage and forecast changes in demand.

The HBP has created six Platform-specific Technology Coordinator roles and one central Technology Coordinator. These are allocated to a specific partner in the workplan of a given platform SP, but can also be assigned to one or alternates, by SP leadership. This Technology Coordination group is responsible for:

1) collecting prioritised use cases from SPs and Co-Design Projects (CDP);
2) guiding the project in prioritising use cases into a Platform-specific roadmap;
3) coordinating technical implementation of the Platform roadmap and managing any required adjustments on the basis of implementation progress and user feedback;
4) planning resources for their respective Platform development, support, and maintenance, while ensuring alignment where use cases cross multiple Platform boundaries;
5) communicating priorities and planning to users and other Platform teams;
6) communicating dependencies on other Platform components to their respective teams;
7) ensuring the technical direction and technical foundation supports existing developers and attracts new developers;
8) ensuring and improving consistency intra- and inter-Platform with respect to user interfaces (UIs), application programming interfaces (APIs), and documentation, wherever necessary;

The seven Technology Coordinators will form the initial Software Development Committee and Infrastructure Development Committees with additional members invited by the initial group. The charters for the Software Development Committee and Infrastructure Development Committee are included in Annex F and Annex G, respectively. Currently, the Software Development Committee and Infrastructure Development Committee are made up of a mix of SP-appointed technical coordination staff and invited members with strong software or infrastructure backgrounds from throughout the HBP.

4. HBP Agile Practice

One the goals of the HBP regarding its information and communication technology (ICT) Platforms is "to demonstrate how the Platforms could be used to produce immediately valuable outputs for neuroscience, medicine and computing". To achieve this goal, the Platform teams must produce immediately useful software for their scientific customers, taking into account the classic project management trade-off triangle of quality, cost and delivery speed.

Because many of the requirements for scientific tools are not known ahead of time, building the right software can only be achieved cost-effectively through close collaboration between software development teams and scientific customers. This challenges classical “big design first” software engineering process models.

The Agile movement grew out of dissatisfaction with existing software development methodologies and attempts to provide a viable approach for the cost-effective delivery of complex and risky software projects. The principles are captured in the Agile Manifesto:[1]

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation;
- responding to change over following a plan.

There are many methodologies that are built on the foregoing principles. Scrum, Kanban and Extreme Programming have been used with great success in many organisations. Each has a well-established discipline, with clearly defined strategies for adoption and adaptation. For reasons of team experience and alignment with current team structure, many teams in the HBP have adopted a Scrum variant. These include the Collaboratory, Brain Simulation, Neuroinformatics, and Neurorobotics teams. It is clear that Scrum favours co-location of teams. As a result, some teams in the HBP have adopted looser Agile variants due to differences in team experience and their geographically distributed nature. Other established teams with collaboration practices which precede HBP are continuing with the practices they are familiar with.

4.1 Scrum: Roles and Procedures

In Scrum, a shippable increment of the software product is produced at the end of each Iteration (also known as a Sprint) for Users to use or test. The Iteration or Sprint is the basic unit of development in Scrum. The duration for each Sprint is fixed in advance, normally between 1 and 4 weeks (typically 2 weeks). There is a Sprint-planning meeting for identifying and estimating Tasks at the beginning and a Sprint review meeting for progress monitoring and future prospects at the end of each Sprint.

There are three roles in Scrum:

1) The Product Owner represents Users as well as Stakeholders.
2) The Development Team is in charge of delivering the Platform in terms of potentially shippable increments.
3) The Scrum Master ensures that the team can do its work without impediment and oversees the development process.

At the beginning of the development, the Product Owner, with the help of the team, transforms Use Cases into small increments called User Stories and containers of Stories called Epics. Epics are thematically organised and usually do not cover a complete Use Case. Epics are collected in the product Backlog, which can be considered as an ordered list of planned activities.

The Product Owner prioritises each Story in consultation with Stakeholders and Users. At the beginning of each Iteration, the team looks at the Stories with the highest priority and decides which of them can be implemented in the Iteration. At the end of each Iteration, the team demonstrates to the Product Owner and Users that the Stories are completely implemented (tested and documented) — see Figure 1.

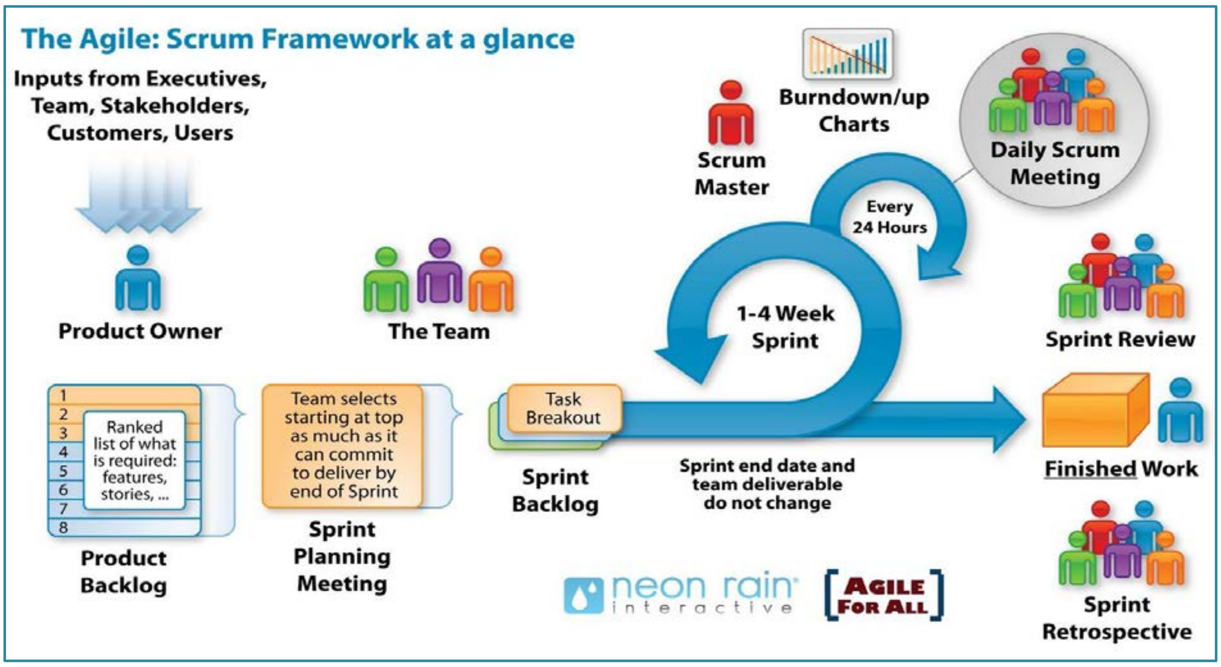

The Agile: Scrum Framework at a glance
Inputs from Executives, Team, Stakeholders, Customers, Users
Product Owner
The Team
Scrum Master
Burndown/up Charts
Daily Scrum Meeting
Every 24 Hours
1-4 Week Sprint
Sprint Review
1 2 3 4 5 6 7 8
Ranked list of what is required: features, stories, ...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Sprint end date and team deliverable do not change
Finished Work
Sprint Retrospective
neon rain interactive
Agile For All

Figure 1: Agile Scrum Iteration Workflow: from the Product Owner Definition of the Backlog to a Finished Product

The Scrum process model outlined in the foregoing enables high adaptability to User needs. It is also inherently able to deal with changes in User requirements: Let one User realise, for example, that a new feature is required that was not thought of at the beginning. By giving feedback, the Product Owner can write new stories and will (re-)prioritise them with all Stakeholders. The key point is that, regardless of the feedback, the Product Owner can take quick concrete actions in order to prioritise User needs.

4.1.1 Scrum Review

One of the goals of working with small Iterations is to have quicker feedback from Users than with a longer release cycle. At the end of each Iteration, a working Platform is released along with a description of the features that have been implemented during the Iteration. This process allowed Users as well as Stakeholders to see the software capabilities at any time during a given SGA.

Gathering feedback from Users on small Iterations is a difficult but rewarding task. In a first step, before the Platform reaches the Minimum Viable Product (MVP) state, internal Users have to test a subset of features. These Users are people involved in the HBP Consortium. Once the MVP state is reached, more Users begin testing the Platform. Throughout the duration of the development, the Stakeholders and team ensure that enough advertisement of the Platform is done in order to attract beta Users.

User feedback is addressed during each Iteration during Backlog discussion between Stakeholders and the Product Owner. Integrating this feedback is vital for the Platform. It reduces the risk of the final delivery failing to correspond to User expectations (see Figure 2).

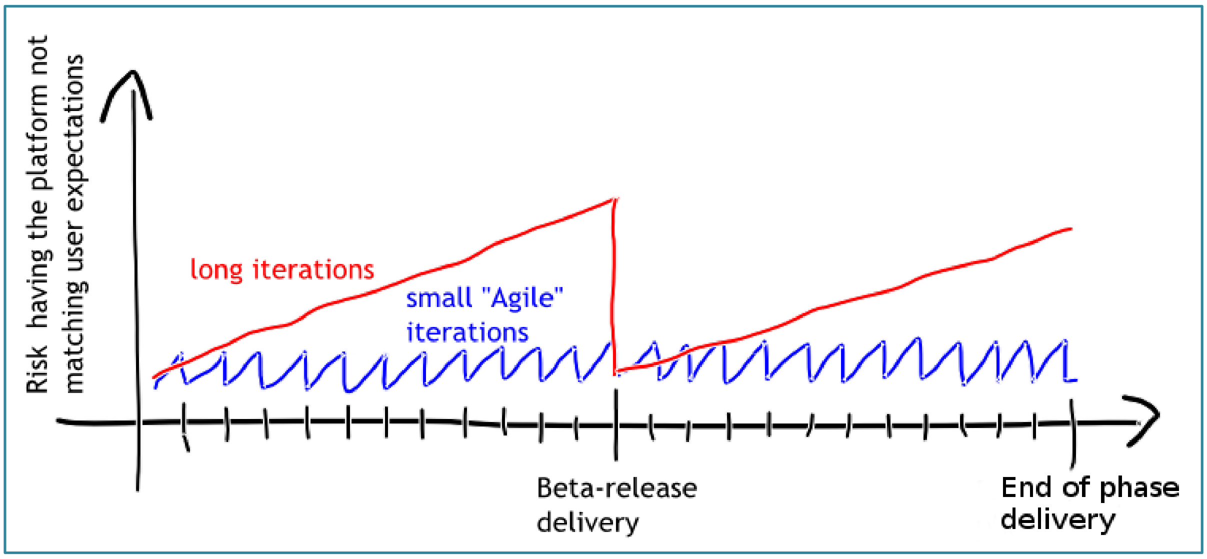


Figure 2: Risk Reduction using a Short Feedback Loop with Users

For every Iteration, the Development Team presents the new features developed during the Iteration in a demonstration to which Stakeholders and Users are invited.

4.1.2 Backlog

In the Scrum methodology, the Backlog usually contains User Stories that deliver User-visible functionality. However, Backlog items can also be bugs that need to be fixed or tasks to be performed. Every Backlog item has an effort estimate and a priority. Backlog items are grouped into Epics that are collections of stories needed to deliver a larger piece of functionality. An Epic in the Brain Simulation Platform might collect all items required to configure and launch simulations from the Collaboratory.

The Backlog is available to Platform Stakeholders at all times. These Platform Stakeholders are expected to be the SP Leader, SP Manager, and the Platform Coordinators. The Backlog is regularly reviewed by the Product Owner and Platform Stakeholders to determine whether some modification to the Backlog contents or priorities have to be made. There are several possible types of modification, as follows:

1) Change of priority in the Backlog, e.g. a User needs a specific feature to demonstrate the software during a conference. The Product Owner, in consultation with Stakeholders, can choose to re-prioritise User Stories related to the feature in order to meet the conference date.

2) Addition of User Stories, e.g. a User needs a feature for his or her project that is not contained in the Backlog. This User can ask the Product Owner to prioritise it.

3) Removal of User Stories: after addition of needed User Stories to the Backlog, the Product Owner realises that the release will not be delivered on time. The Product Owner can decide to change the status of less-needed stories in the release Backlog in order to ensure that the release meets the deadline. These stories can be placed lower down in the request Backlog or dropped if they are no longer valuable for the Platform.

4) Change in the estimation of the stories. The development team regularly re-estimates the stories in the Backlog. It's very difficult to have exact estimations at the beginning of a project. Regular re-estimation improves overall confidence for meeting the deadlines.

Flexibility in Scrum is accompanied by strongly defined team roles and meeting practices. Scrum is well suited to iterative new product development, but there are many cases in which the limited structure impedes the work to be done. In these cases, prevalent in operations and integration-heavy environments, the Kanban Agile variant (see Section 4.2 below) is highly applicable.

4.2 Kanban

Kanban is a methodology originally developed by Toyota in the 1940s to improve the efficiency of their manufacturing and engineering processes. Its first application to knowledge work date back to 2005, with proper formulations emerging in 2009–2010, thanks to the works of David J. Anderson, Jim Benson, Corey Ladas and others. Kanban is easy to integrate with other processes, presenting minimal entry barriers and organisational constraints.

Kanban consists of three main rules and one tool (the Kanban board):

1) Visualise your workflow.
2) Limit your work in progress (WIP).
3) Measure the flow.

These rules are explored in more detail in Sections 4.2.1 to 4.2.3.

4.2.1 Visualise your workflow

The literal translation of the Japanese word *Kanban* is "visual signal". In Kanban, every work item is represented as a separate card on the Kanban board. Physical boards typically use adhesive notes on a whiteboard; online boards draw upon the whiteboard metaphor in a software setting.

The use of cards on the Kanban board provides a way for the team to follow the progress of tasks through its workflow. Kanban cards contain all crucial information about that particular work item and typically give the team visibility into: the team member responsible for the task; a description of the job being done; and an estimate of the duration of the work.

Kanban boards are organised in columns, with each column representing a step in a given workflow. As work progresses on an item, its card is moved from one column to the other (usually from left to right). When the work item is completed, the card is removed from the board.

A basic Kanban board has a three-step workflow: To Do; In Progress; and Done. However, depending on the size, structure, and objectives, the workflow can be mapped to meet the unique processes of any particular team. For simple processes, board columns are usually enough, whereas more complex processes typically combine columns and horizontal lanes (also called *swimlanes*).

Kanban not only visualises work, allowing development teams to observe the flow through the system, but also reveals tasks that are blocked or workflow bottlenecks and queues. Visualisation of workflow can allow a team to improve communication and collaboration.

4.2.2 Limit your work in progress

A number at its top indicates the limit on the number of cards allowed in the column. Work items enter a column only when it has free spots and the team focuses only on the work that is actively in progress.

The limits are the critical difference between a Kanban board and any other visual storyboard.

Limiting the amount of WIP at each step in the process prevents excessive multitasking and reveals bottlenecks dynamically so that the team can address them, e.g. the entire team can swarm on a work item to get the process flowing smoothly again.

Another goal of limiting the amount of WIP is to match workflow to system capacity. In other words, a system can only handle so much traffic moving smoothly through the steps in the process.

In the Kanban mindset, keeping work moving is much more important than keeping workers busy. This concept is usually summed up in the sentence, "Stop starting, start finishing".

4.2.3 Measure the flow

The visualisation of workflow and WIP limits help the team to collect metrics to analyse flow and even to obtain indicators of future problems.

Key metrics in Kanban are as follows.

- *Cycle time* is the amount of time it takes for a unit of work to travel through the team's workflow–from the moment work starts to the moment it ships. By optimising cycle time, the team can confidently forecast the delivery of future work.
- *Lead time* is the amount of time passed between the moment the task is created and when it's completed.
- *Throughput* is the number of work items completed during a given period of time (e.g. 1 day).

Other metrics could be total WIP or the number of blockers. A cumulative flow diagram shows the number of issues in each state and is a useful way to visualise some of these metrics (e.g. see Figure 3, credit Pavel Brodzinski, slide 36 of https://www.slideshare.net/pawelbrodzinski/kanban-basics-5834758).

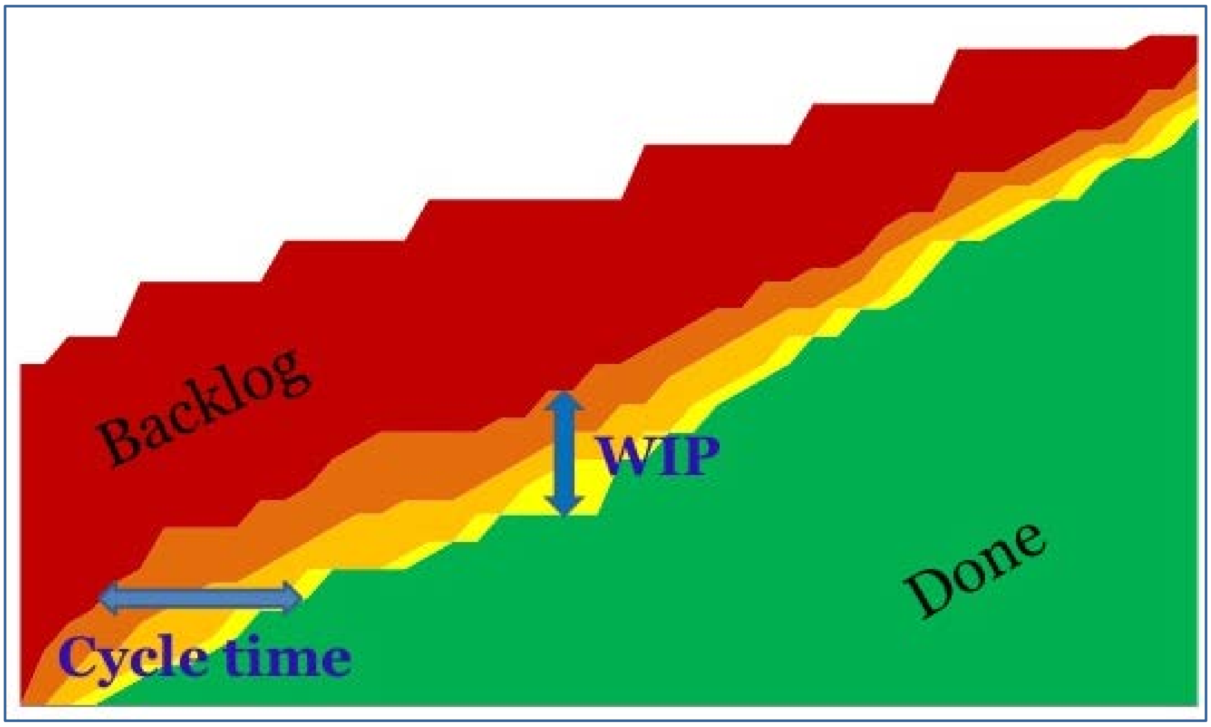


Figure 3: Kanban kick start: Estimation

4.2.4 Kanban: final remarks

The ultimate objective of Kanban methodology is to help teams to improve collaboration and pursue incremental evolutionary change. The Kanban system reveals how work flows through a process and gives the User tools to evaluate workflow and how to improve it. Once the User begins to analyse workflow — and measure things like total WIP, blockers, throughput or lead or cycle time — ways to evolve and streamline become apparent. The methodology advocates a focus on continually improving team efficiency and effectiveness with every iteration of work.

In the words of one of the principal advocates of the Kanban approach:

- "Kanban is not a software development or project management lifecycle or process. … The Kanban Method is an approach to change management". — David Andersen — http://www.scrumcrazy.com/David+Andersen+-+There+is+no+kanban+process+for+software+development.
- "Many results of Kanban are counterintuitive. What appears to be a very mechanical approach — limit WIP and pull work — actually has profound effects on people and how they interact with one another." (Anderson, 2010,[2] p. ix)

4.3 Integration of Larger Projects

The HBP must also consider the case of multi-team integrations. Agile methodologies address larger scale projects with a similarly iterative approach to integration. The approach is to have "Integration Events" which take the place of a standard iteration and involve any component teams needed.

During the Integration Iteration, the combined product is integrated, tested and validated by the combined team, as in a standard iteration. It is best if these integration iterations include only two teams. See Figure 4.

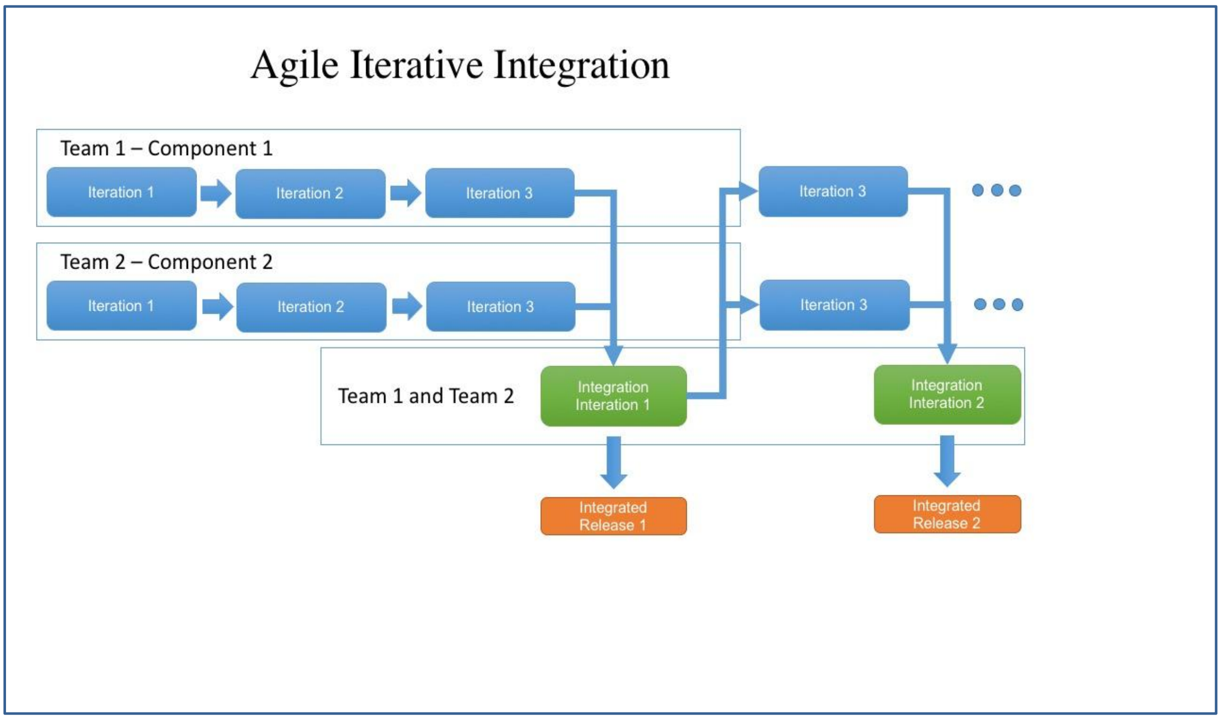


Figure 4: Agile iterative integration

5. Software Engineering

5.1 Software Development

In Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) tiers of the hierarchy, software is a crucial component. All research activities in the HBP are heavy consumers or producers of complex software packages. A robust approach to software quality will therefore be a critical factor in the success of the HBP.

Drawing on experience of other large-scale software projects, the HBP has identified the need to invest in software maturity at an early stage.

These investments have continued and expanded to include the following best practices:

- version control — all software in the HBP must be developed using a version control system;
- unit testing, typically with minimum test coverages (e.g. 90% or higher on all Collaboratory and Brain Simulation Platform Team developments);
- automated integration testing suites (automated post-deployment system testing);
- continuous integration;
- ticket management for project feature planning;
- scrum training for Agile software development (either with an Agile coach or an HBP practitioner);
- coding standards;
- utilisation of code review by both Developer and Scientific Developer teams.

Bringing all these pieces together results in DevOps. All teams in HBP use a variant of this approach with slightly different mixes of tools.

5.2 Operations Standards (DevOps)

In future phases (SGA2 and beyond), the HBP may invest in a dedicated, centrally supported DevOps stack that includes standardised tooling for the app development lifecycle. The metrics gathered by such a system could be the backbone on which a central quality assurance (QA) effort would be built to meet the criteria for the HBP Managed Infrastructure Tier. It expected that a number of the components below will be provided by SP7's Fenix initiative (previously delivered as FeDaPP Architecture Report V01.docx, for the Oct 2016 DPIT review).

There are four main types of infrastructure that are needed to support the efficient development and operations of Service Oriented Architecture. They are:

1) development tools: Source control, Issue/Requirements/Backlog tracking, Code review tools, Continuous integration and Continuous deployment;
2) PaaS — proxy servers, log collection and monitoring services, databases;
3) IaaS — virtual machines (VMs), storage, network and network configurations;
4) HPC — high-performance computing (HPC) storage (General Parallel File System, GPFS), networks compute clusters and supercomputers.

Development tools are crucial to maintaining a consistent flow of tracked code and configuration changes, as well as ensuring that feature updates are tested and deployed with minimum manual effort. This process, currently in development and planned to be deployed at the end of SGA1, is outlined in Figure 5.

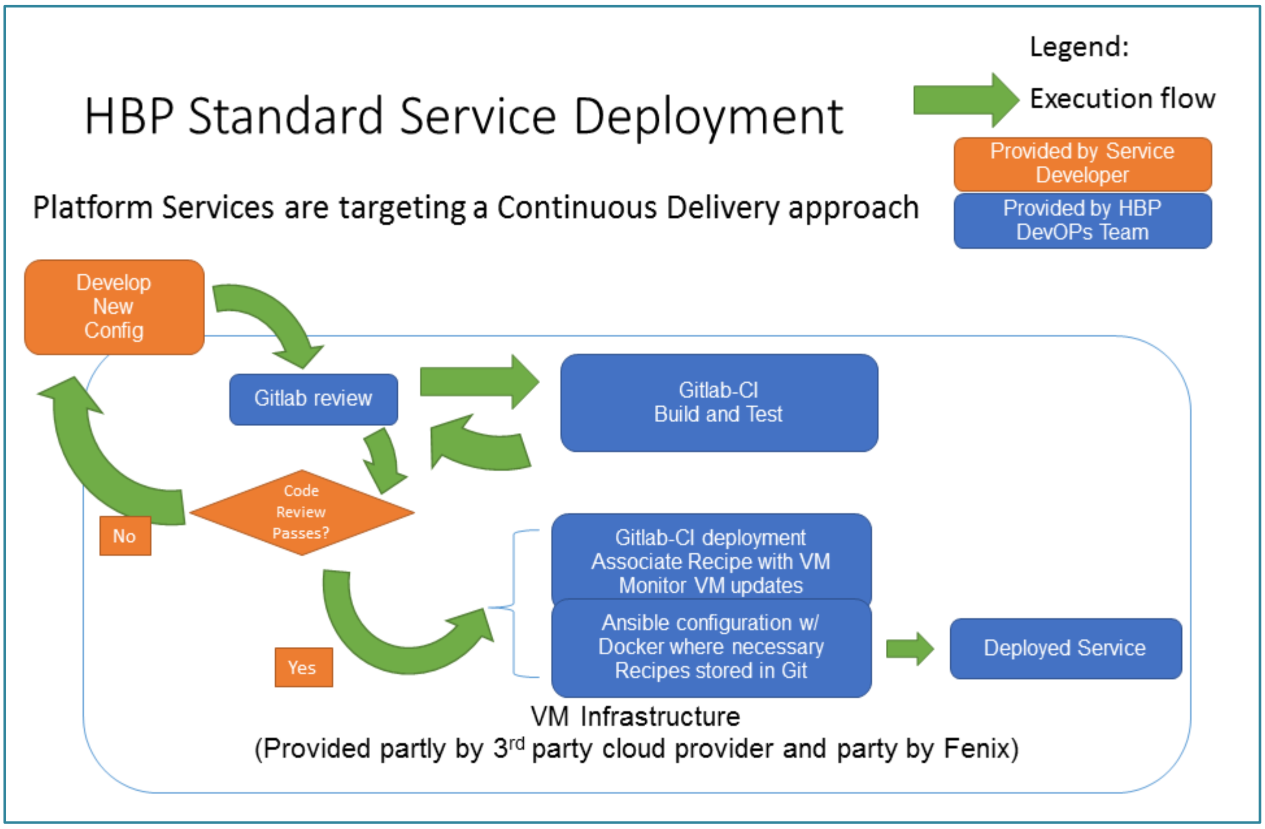

HBP Standard Service Deployment
Platform Services are targeting a Continuous Delivery approach
Legend:
Execution flow
Provided by Service Developer
Provided by HBP DevOPs Team
Develop New Config
Gitlab review
Gitlab-CI Build and Test
Code Review Passes?
No
Yes
Gitlab-CI deployment
Associate Recipe with VM
Monitor VM updates
Ansible configuration w/ Docker where necessary
Recipes stored in Git
Deployed Service
VM Infrastructure
(Provided partly by 3rd party cloud provider and party by Fenix)

Figure 5: HBP standard service deployment

For the SaaS, PaaS and IaaS tiers, reliable and repeatable deployment software is a crucial component of the overall operations strategy. Monitoring of services will be based on best practices from industry along with crucial expertise from HPC providers in the HBP and federated infrastructure providers throughout the EU.

HPC centres (and by extension SP7) have well-defined deployment and monitoring practices that are informed by a long history of service provision. For the other Platform teams, no such history exists, so best practices have been adopted from Agile software development. Platform teams in the HBP have been adapting these techniques and tools according to their team needs.

To this end, a DevOps model is in the process of being pragmatically adopted across HBP as follows.

- Deployment lifecycle with development, staging (optional), and production environments for all services.
- Continuous integration — unit testing, integration testing, repeatable software builds and package releases.
- VM configuration development — source control and code review of configuration changes. Requires a programmable configuration system.
- VM configuration management — associate service configurations with specific VM resources.
- VM configuration deployment — deploy approved changes through an automated system.
- Object Storage — highly available, redundant storage for VMs and service data.
- Internet Gateway with Caching Proxy Server — services typically are not available directly via the internet. Best practice places a tuned caching proxy server between application servers and the open internet for reliability, flexibility, and performance reasons.

While the service classes are mostly common to the various Platforms, variation exists, and there are special needs in some Platforms. As the various capabilities of the Platforms were developed initially in the Ramp-Up phase and with the subsidiary principle in place, there is substantial heterogeneity between Platforms. This reflects not only the different times when specific capabilities were deemed necessary, but also the subsidiary principle of the HBP at work. The SGA1 Description of Actions makes no provision for activities to fundamentally homogenise DevOps practices.

5.3 HBP Standard DevOps Stack

As this collection of services is being standardised and brought into service, it will be initially tested by the HBP Collaboratory and any other interested Platforms. At the time of drafting, the HBP Standard DevOps Stack is in use for selected services in:

- the HBP Collaboratory;
- SP5 Neuroinformatics Platform.

Table 1: HBP Standard DevOps Stack

Service Category	Service	Provided by	Notes
Dev hosts	Openstack	ETHZ-CSCS under Fenix initiative	
Staging hosts	Openstack	ETHZ-CSCS under Fenix initiative	Optional environment
Production hosts	Openstack	ETHZ-CSCS under Fenix initiative	
Continuous integration	Gitlab	Collaboratory DevOps	
VM configuration development	Git	Collaboratory DevOps	
VM configuration management	Ansible and some Docker	Collaboratory DevOps	
VM configuration deployment	Ansible and some Docker	Collaboratory DevOps	
Object Storage	Ceph	ETHZ-CSCS under Fenix initiative	
Internet Gateway with Caching Proxy server	NGINX	Collaboratory DevOps	
Monitoring	Zabbix, Elasticsearch, Kibana, telegraph	For SP5 services: Collaboratory DevOps For Fenix services: joint effort of HPC sites	
Runtime Profiling	Pinpoint (Java), Datadog (Python)		

User support tickets	TBD	For HBP Platforms: Tier 0 – Collaboratory Platform Support For Fenix services: BSC, with support of all HPC sites	Currently handled by support email with internal ticketing service
Container services	Docker	Collaboratory DevOps	
Release software repository	Github		
Database service	MySQL, Postgresql	Collaboratory DevOps	

5.4 BBP Standard DevOps Stack

At the time of drafting, the Blue Brain Project (BBP) Standard DevOps Stack is the most widely used of the DevOps infrastructures. The BBP Standard was developed by various teams at the BBP and serves as the basis for significant parts of the development, deployment, configuration, and monitoring of the following Platforms:

- Collaboratory;
- SP5 — Neuroinformatics;
- SP6 — Brain Simulation Platform;
- SP10 — Neurorobotics Platform.

Table 2: BBP Standard DevOps Stack

Service Category	Service	Provided by	Notes
Dev hosts	Openstack	BBP Core Services	
Staging hosts	Openstack	BBP Core Services	Optional environment
Production hosts	VMWare	BBP Core Services	
Continuous integration	Jenkins	BBP Core Services	Jenkins uses Openstack VMs for jobs
VM configuration development	Git + Gerrit	BBP Core Services	
VM configuration management	Foreman	BBP Core Services	Integrates with VMware and Openstack
VM configuration deployment	Puppet	BBP Core Services	
Object Storage	Ceph	BBP Core Services	Used by both Openstack VMs and Collaboratory Storage Service
Internet Gateway with Caching Proxy server	Apache Traffic Server	BBP Core Services	

Monitoring	Icinga, Grafana, Kibana, syslogd, collectd	BBP Core Services	Only usable by BBP or affiliated users
User support tickets	JIRA	BBP Core Services	Only usable by BBP or affiliated users
Container services	Docker	BBP Core Services	
Release software repository	Nexus, internal Pypi proxy, and bower repositories		Only available inside the EPFL network
Database service	Postgres and Mysql DBaaS	BBP Core Services	Only available inside the EPFL network

5.5 Medical Informatics Platform Standard DevOps Stack

CHUV has developed its own DevOps approach due to the lack of a centrally supported standard. The CHUV standard approach is tailored to the realities of operating in a hospital IT environment. This system forms the foundation of development activities in SP8, Medical Informatics.

Table 3: Medical Informatics Platform Standard DevOps Stack

Service Category	Service	Provided by	Notes
Dev hosts	Docker	Individual developers	
Staging hosts	Docker, Mesos	CHUV	Staging can also work inside Vagrant VMs on a developer desktop
Production hosts	Docker, Mesos	CHUV	
Continuous integration	Jenkins, Travis-ci.org, CircleCI.com, Werker.com, Codacy.com	CHUV, free cloud services	
VM configuration development	Git, Github, BitBucket	github.com, BitBucket.com	BitBucket is used to store confidential and encrypted information for the deployments on each hospital
VM configuration management	Git and Ansible	CHUV	Template project:[3] adapted from Cisco Mantl.io project
VM configuration deployment	Docker, Mesos, Marathon and Ansible	CHUV	

Object Storage	NA		
Internet Gateway with Caching Proxy server	NGINX	CHUV	
Monitoring	Collectd, Marathon, Consul, StatusCake,[4] RunStatus[5]	CHUV, EPFL	
Container services	Private Docker repository, Docker hub	CHUV, docker.com	
Java/Maven jars repository	Artifactory	CHUV	
Python package repository	Github		installation from source with pip
Bower repository	NA		
Database service	Postgres and RAW DBaaS	CHUV, EPFL	Deployed by Ansible on a Mesos cluster

For the Medical Informatics Platforms, the following tools are used for project management:

- Codacy.com
- Waffle.io
- YouTrack (Jetbrains)
- Trello.com

In addition, our projects aim to follow a common development process.[6]

Finally, a catalogue of HBP software is used also as a dashboard to monitor the overall health of the SP.[7]

5.6 Neuromorphic Platform Standard DevOps Stack

The Neuromorphic Platform has developed its own DevOps approach due to the lack of a centrally supported standard.

5.6.1 Job Queue and Neuromorphic Collaboratory Integration

Table 4: Job Queue and Neuromorphic Collaboratory Integration

Service Category	Service	Provided by	Notes
Dev hosts	Docker	individual developers	

Staging hosts	Docker	public Cloud via CNRS	
Production hosts	Docker	public Cloud via CNRS	current provider is Digital Ocean
Continuous integration	Travis CI	Travis CI	
VM configuration development	Git / Github	github.com	
VM configuration management	Docker Hub / in-house Python application	docker.com, CNRS	
VM configuration deployment	In-house Python application	CNRS	
Object Storage	Collab Storage	Collaboratory	
Internet Gateway with Caching Proxy server	–	–	not currently implemented
Monitoring	Digital Ocean (hardware) StatusCake (services)	Digital Ocean StatusCake	
Container services	Docker Hub	docker.com	
Python package repository	Pypi (if needed)		
Bower repository	Not used		
Database service	Postgres	public Cloud via CNRS	

5.6.2 SpiNNaker Large Machine Service

Table 5: SpiNNaker Large Machine Service

Service Category	Service	Provided by	Notes
Dev hosts	Maven Jetty	Individual developers	
Staging hosts	Maven Jetty	Local VM Server at Manchester	
Production hosts	Maven Jetty	Local VM Server at Manchester	
Continuous integration	Travis CI	Travis CI	
VM configuration development	Xen VM	Local VM Server at Manchester	Some parts need to be backed up and documented

VM configuration management	—	—	Not currently implemented
VM configuration deployment	—	—	Not currently implemented
Object Storage	RAID-10 and RAID-6 array backed up to University Central Storage	Local VM Server at Manchester / U of Manchester	
Internet Gateway with Caching Proxy server	—	—	Backend Service — not relevant
Monitoring	—	—	Not currently implemented
Container services	Not used		
Release package repository	git		
Database service			Backend Service

5.6.3 *NMPM-1 (20-wafer BrainScaleS System) Service*

Table 6: NMPM-1 (20-wafer BrainScaleS System) Service

Service Category	Service	Provided by	Notes
Dev hosts	Custom (SLURM, LXC, build flow)	locally	
Staging hosts	Custom (SLURM, LXC, build flow)	locally	
Production hosts	Custom (SLURM, LXC, build flow)	locally	
Continuous integration	Jenkis CI	Jenkins server running locally	
VM configuration development	Versioning: Git (reviewed via Gerrit); development: manually/low-level	—	
VM configuration management	Salt/Git	—	
VM configuration deployment	Salt	—	
Object Storage	NFS (RAID-10 and RAID-6 array, partially backed up to University Central Storage)	locally	

Internet Gateway with Caching Proxy server	—	—	Backend Service
Monitoring	Carbon/Graphite (fed by custom services), syslog, snmp	locally	
User support tickets	Redmine	locally	
Container services	Shifter		
Python package repository	Spack	github/locally	Some packages still maintained locally
Bower repository	—	—	Backend Service
Database service	—	—	Backend Service

5.7 HPAC Platform Standard DevOps Stack

The software developed in SP7 is not a single software stack, but a large variety of independent tools and frameworks, which are to a large extent based on previous projects. The quality and testing strategies for software and services are described in detail in Annex H of Deliverable D7.7.5 (Ramp-Up Phase).[8]

5.8 Long-term software maintenance

The HBP has studied guidelines for sustainable scientific software, such as the TriBits model[9] and numerous HBP participants have extensive experience in the initiation and execution of long-running community driven scientific software projects, e.g. NEST, PyNN, Neo, and NEURON. For those projects that are not suitable as sustainable open-source packages, the HBP will endeavour to transition to infrastructure programmes to ensure long-term funding for its scientific software development activities.

6. Quality Assurance

Standard Agile practice requires that the discipline of rigorous testing and user validation is owned and performed by the feature teams. In HBP, this leads to several important principles:

1) Making project value understood by Platform teams;

2) Use of user validation as key to ensuring that the right use cases are prioritised;

3) Effective and efficient testing as a cornerstone of quality software.

In the case of Principle 1), two factors are key. Lean management emphasises the importance of the “definition of good” being clearly understood by implementation teams. Does “good” mean, for example, that a certain analysis can be performed more predictably in the same time or does it mean that the same analysis must be performed in one-tenth of the current time. To resolve this ambiguity, the definition of good must also include the need to accelerate ongoing science. Principle 1 drives quality pragmatically and principle 2 prioritises immediately valuable feature development. Principle 2 has the additional benefit that it should only drive optimisations where they are really needed, rather than based on some vague theoretical model of what is needed.

In practical terms, Principle 2) is addressed by involving HBP users in the validation and verification of the Platforms wherever feasible. This has been ongoing since the Alpha releases in M18–M24 of the Ramp-up Phase and is expected to continue for the duration of the HBP. For example, Platform development teams have been tightly integrated into Collaboratory feature prioritisation since the initial release at the end of April 2015.

Platforms have adopted a user-first approach to validation of use cases and user stories where the user is explicitly a part of the process for planning of development priorities and validating development products. Internal users in the Platform SPs were among the first users of the Platform tools. Additionally, courses offered since the Ramp-Up Phase by the HBP Education Programme and by HBP Partners engage students in HBP next-generation tools. Additionally, plans for SGA2 are in place to incentivise external high-level scientific collaborations with targeted high-level support allowing for deep customisation and steering of Platform use cases by pairing scientific and engineering expert support resources inside the project with an allocation system called the High Level Support Team, as outlined in WP5.9 of the SGA2 Proposal.

Principle 3) is key for a software-driven ecosystem like the HBP Platforms and the HBP Collaboratory, software QA must be undertaken *strategically* and it is clear that systematic testing practices play a crucial role. Depending on the targeted maturity of a given piece of software, different practices can be employed because many QA practices imply potentially higher short-term costs. However, it is also clear that specific development techniques, for example Test-Driven Development (TDD), yield a multitude of benefits as they enable the team to catch errors early and develop features more aggressively, regardless of maturity. The value of early detection and correction of errors and the resulting cost savings are well-studied phenomena.[10] As a result, rigorous multi-level testing approaches are recommended for any long-term infrastructure software or service project in the HBP.

Testing approaches are applied by the HBP across numerous phases of development and the application of these phases in the various Platforms is described in Sections 6.1 to 6.6. The selection and application of the specific approaches is delegated to the Platforms in accordance with the subsidiarity principle described in the Principles of this document.

6.1 Unit Testing

Unit testing consists of decomposing software functionality into functions and class units and testing their individual functions independently from the software system of which they are part. This gives increased confidence about the function of the individual components, but is seen by some to require additional investment during development. However, in most HBP teams involved in Platform development, the value of investments in repeatable testing is understood. The long-term benefits of such investments are accepted and unit testing is widely used.

6.2 Integration Testing

In addition to unit testing, investment in integration testing, in which various system components are tested together, is also recommended. These tests can find a multitude of uses from post-modification system correctness testing during development, system correctness testing in early integrated “dev” environments in a continuous integration system to provision of a subset of system health metrics for part of a service monitoring system.

6.3 Manual Testing

Manual testing is typically done on graphical user interfaces when automated testing is more challenging to set up and maintain. Additionally, manual testing plays a key role in the testing of scientific software. HBP scientists using software tools play an essential role in manual testing of these tools. These scientists are the principal experts in the application of appropriate software tools and often have the best understanding of their correct behaviour; consequently, they are able to detect deviations. In combination with continuous integration testing, this approach has been used successfully in Neural Simulation Technology (NEST) development for nearly two decades.

6.4 User Interface Testing

User testing practices are fundamental to Agile software development. Wherever possible, user validation is done during each iteration. In the development of new user-visible features, the development teams also perform user testing. This requires that the development teams have developed an understanding of user needs and engage in regular interactions with potential customers. This is possible in part because all Platform SPs are structured to include scientific Tasks that are consumers of the respective Platforms. Coupled with Agile software development, SPs that engage their internal audiences are able to get early validation of their development plans, even involving them in prioritising the Backlogs. This can be seen in the processes and examples outlined in Annex E.

6.5 Monitoring

Monitoring can be seen as continuous automating testing of specific service characteristics and is considered mandatory for any service targeting the HBP-defined internal criteria for technology readiness level (TRL) ≥7 (see Annex A). It should cover some of the same behaviour as integration testing, but must be done in a fashion that does not degrade service behaviour and performance, while still attempting to continuously test the service in question.

6.6 Platform-testing specifics

6.6.1 Collaboratory, Brain Simulation Platform, Neuroinformatics and Neurorobotics

Testing standards used in development are described here in Annex C and Annex D. With some variation in specific practices, this process is used by the Collaboratory, Brain Simulation Platform, Neuroinformatics Platform, and Neurorobotics Platform.

Notably, the Collaboratory and Neuroinformatics Platform are in the process of migrating from the BBP testing infrastructure to the HBP Standard DevOps stack described in Section 5.3. However, it expected that testing standards will remain largely the same otherwise.

Specifics of user interface (UI) testing are described in Annex E with examples from the Collaboratory, Neurorobotics Platform, and Medical informatics Platform.

6.6.2 Neurorobotics

The Neurorobotics Platform SP (NRP) has a Task T10.6.2 dedicated to “testing, profiling and quality assurance”. At the time of drafting, the most recent version of D10.7.2 provides examples of NRP monitoring, analytics and testing activities.

6.6.3 Medical Informatics

Open-source projects on the Medical Informatics Platform should be hosted on Github, which grants access to a free set of tools for continuous integration, software quality control, code coverage, dependency analysis, and project management. The Medical Informatics Platform is currently also using the following services as part of its testing infrastructure:

- Travis-ci.org
- CircleCI.com
- Wercker.com

Different continuous integration tools allow different aspects of SP8’s services to be monitored: code quality, compilation and execution of unit and integration tests, installation of HBP software components on a VM.

Specifics of UI testing are described in the Annex E with examples from the Collaboratory, Neurorobotics Platform, and Medical informatics Platform.

6.6.4 Neuromorphic Testing

When compared with the other HBP Platforms, the Neuromorphic Platform has an additional facet to consider when it comes to testing. Because they develop hardware as well as software and services, they must also consider how their custom hardware solutions behave under a multitude of conditions and workloads. Examples of such practices are described in the D9.5.1 and are not repeated in detail here.

Notably the following Tasks have a role to play in the testing of the hardware aspects of the Platform:

T9.1.3 – Benchmarking with the SNABSuite benchmarking framework;

T9.2.2 – Hardware testing, in particular the HICANN testing module example;

T9.2.4 – Small Scale Simulator equipment for testing.

6.6.5 HPAC Platform Standard

Due to the length of time spent in development and the wide variety of software developed in SP7, HPC software development practices are less standardised than in some of the newer software Platforms in the HBP. However, as the context is high-end computing, HPC Centres (and by extension SP7) follow rigorous practices in software development, tailored to the particular community served.

For example, NEST Development in SP7 and SP6 is guided by developer guidelines.[11] NEST development has been using continuous integration testing since 2011. Since May 2015, CI testing is based on Travis,[12] testing all pull requests to the NEST master repository.[13] Testing comprises code style checks, static code analysis and a comprehensive test-suite combining unit and integration tests. Automated testing is complemented by manual code review in the Github repository, with merge privileges restricted to a small group of experienced developers.

7. Recommendations

The following recommendations are designed to address two major issues related to quality. The first is uniformity of communication regarding software, services, and workflows developed by the project. The second is a high-level view on QA with clear recommendations on Quality Ownership and Quality Checks. The Quality Assurance section (7.2) includes recommendations on output artefacts for the two major styles of development in the Project Lifecycle.

It should be clear from the description of HBP Managed and HBP Coordinated Tiers in the FPA and SGA1 that these recommendations are to be considered mandatory for the HBP Managed Tier and strongly recommended for the HBP Coordinated Tier. The exact definition of components in the HBP Managed Tier is ongoing and the first version is expected to be provided at the end of SGA1 with the definition of the HBP Infrastructure Plan MS11.3.4.

7.1 Communication

The work of the Software and Infrastructure development teams in the HBP is focused on delivering software and services. However, clear communication is key to ensuring that these product offerings meet user expectations.

One facet of communication is that the audience for a product should be clearly listed. An immature product should be labelled as such and a mature product must have undergone adequate testing and validation.

Consequently, it is recommended that all HBP software or service components label their target audience and maturity level consistently so that users can know what to expect.

The labels below are intended to be attached to application catalogues, software catalogue entries, Collaboratory documentation, Github READMEs. If you need to include these labels, please contact the HBP Collaboratory team for the latest style guide, labelling guidelines, and icon library.

The communication recommendations in sections 7.1.1 to 7.1.3 largely follow the classifications initiated by the Brain Simulation Platform in their March 2016 Platform release. The major change is that the definitions have been generalised to make them applicable to any software, services or workflows developed by the project.

7.1.1 Audience

Everybody

Everyone interested in using the HBP Joint Platform infrastructure and facilities in the easiest way for relatively simple collaborative scientific projects using GUIs or simple, well-documented Jupyter notebooks. These tools must be accessible to users with expertise in a particular scientific domain, but with little or no background in computer programming, simulation or image processing tools. Examples might be a clinician who uses MRI images as a diagnostic tool or an electrophysiologist who deeply understands cell physiology, but has no exposure to Python programming.

Power users

Power users are interested in using some of the deeper capabilities of the HBP Joint Platform for collaborative projects. They are able to design, implement, and run computational pipelines to analyse volumetric data, models or simulations. They have a working knowledge of at least one programming language, likely Python.

Experts

Experts and co-design partners are users who are experts in one or more scientific domains. They may also have substantial expertise in implementing custom solutions with complex software systems. They are early scientific adopters of key software, service and workflows, and have a hand in providing deep design feedback early in the development process. Usage by users outside HBP should be invitation only and should be paired with extensive support, either from the component developer or from the High-level Support Team.

Developers

Software developers are designers, implementers and early adopters of initial versions of software, services, and workflows. Components may be acceptable for limited release to audiences of software experts.

7.1.2 Component maturity

If a component does not have a maturity label, it is expected to be of production quality. This level corresponds to TRL7–9 and the component is expected to be highly robust and usable by the intended audience. Mature software need not be easy to use, if it targets Power users or Developers, but it must still be well documented and predictable in its error handling and performance profile.

Icon to be determined

Production: A service or software of this maturity level has reached a high-level robustness and may be used by a wide audience with high levels of confidence. The target audience may be End users, Power users, Experts or Developers. Production tools which target Power users or Developers may not be easy to use, but they will be well documented, well tested, and can be expected to fail gracefully in documented ways. Support channels must be well established.

Icon to be determined

Beta: A service of this maturity level has reached a certain robustness and may be used by early adopters. This level corresponds to TRL5–6. The target audience may be End users, Power users, Experts or Developers.

Icon to be determined

Experimental: A service of this maturity level is under heavy development and is recommended only for specialist use or use for co-design partners. This level corresponds to TRL3–4. Generally, these components should also be tagged as intended for Expert or Developer audiences.

7.1.3 Compute Requirements

The compute requirements label applies specifically to HBP-provided services or workflows. If no special compute requirements exist for a service or workflow, or if the compute requirements are completely handled by the service in question, they should be labelled "without compute requirements". However, in some cases, additional user steps are required to allocate desktop, HPC or cloud resources to use a tool. In these cases, the tool should be labelled accordingly.

Icon to be determined

Desktop: Software, services or use cases with this icon can be run on your laptop or desktop without a requirement for high-powered computing resources.

Icon to be determined

HPC: Software, services or use cases with this icon require High-Performance Computing resources. They can be either public, such as those available through the Network Services Group, or provided by the user through a personal grant, such as a Partnership for Advanced Computing in Europe (PRACE) award granted from one of the supercomputer centres supporting HBP activities.

Icon to be determined

Bring Your Own: Software, services, and use cases with this icon need substantial compute resources from outside the HBP. Subject to technical compatibility, these products will require access to a large compute allocation through a personal grant or a user allocation of compute resources with a cloud provider.

Icon to be determined

Neuromorphic: This tool has a requirement for neuromorphic computing resources. These are accessible from the Neuromorphic Computing Platform and may be in the form of access to the neuromorphic job queue service or through locally installed neuromorphic computing systems.

7.2 Quality assurance

The recommendations for QA depend on the development methodology that the component team is following. However, teams running an Agile process without adherence to the Agile QA practices (see section 7.2.1) may need to fall back on Checkpoint QA (see section 7.2.2) if appropriate QA activities are not included in their iterative development process. It is also important that when SPs report to the European Commission on component releases, they report the QA process they are using and their QA owner for that release. Finally, it is possible for QA responsibilities to be assigned in different ways and for a Platform QA process to differ significantly from the recommendations given in this section. However, in case of significant deviations, the QA process should be documented as part of the component documentation.

7.2.1 Agile Quality Assurance

Agile QA can be described as a continuous multi-level process integrating comprehensive testing and user feedback throughout the iterative development process. The quality checks listed are recommended, but some deviation is allowed if the product is in the early stages of development or lower maturity levels are targeted. After initial release, it is recommended that all the Quality checks be performed on each iteration (for SCRUM) or on each feature release (Kanban).

For Agile QA, the checks below should verified by the QA Owner; alternatively, the QA owner needs to create a priority ticket for the next iteration to resolve the issue. While the QA owner verifies that the checks are happening, the actual checks should be done by the component development team themselves as part of the User Storage Acceptance Criteria or a more general Definition of Done for the team. Deferring the implementation of such tickets incurs so-called Technical Debt and is considered a failure of the Agile QA process. In the event of repeated failure of the Agile QA process, the QA Owner should schedule QA checks (see the list below) to restore confidence in all facets of component quality.

QA Owner:

Recommended to be the Product Owner for the component.

QA Checks:

1) Unit test coverage is set to a minimum of 80%.
2) Integration tests are added with each new feature added.
3) Regressions are fixed with highest priority (immediately). This can be a requirement in the Definition of Done.
4) Bug reports are followed by new integration or feature tests if possible.
5) Major UI changes, API revisions, schema or format revisions are mocked-up and tested with users prior to development.
6) New or enhanced features/user stories are user tested before or immediately after release.
7) Code changes are reviewed.
8) User and developer documentation is updated with each feature release and is reviewed.
9) Services should have monitoring systems to verify that basic functions are continuously operational. These should be designed to monitor the service level agreement (SLA) variables at 7 or higher.

7.2.2 Checkpoint Quality Assurance

Checkpoint QA is similar in principle to Agile QA with larger gaps between QA checkpoints. This necessitates larger QA efforts at the checkpoint, but alleviates some of the integrating comprehensive testing and user feedback throughout iterative developments. The quality checks listed below are recommended, but some deviation is allowed if lower maturity levels are targeted.

NOTE: Quality checkpoints should be no more than 6 months apart.

QA Owner:

Recommended to be the Owner for the component.

Quality Checks:

1) Use cases for new or enhanced features are documented.
2) UI features, API, schema or format changes are mocked-up and tested with users prior to development. The resulting feedback is integrated into the Test Plan to ensure that the final product matches user expectations.
3) Quality criteria documented since the last Quality Checkpoint have been added to the Test Plan.
4) Unit test coverage is set to a minimum of 80%.
5) One or more integration tests are added with each new feature added since the last checkpoint.
6) Bug reports are followed by new integration or feature tests if possible.
7) New or enhanced features/user stories are tested prior to release according to the documented Test Plan.
8) User and developer documentation is updated with each component release and is reviewed by a team member who is not the author.

As a result, it is expected that Checkpoint Quality Assurance follows these phases and upon completion of a given phase, produces the following artefacts:

1) Phase: Specification
    a) Use cases, functional and non-functional requirements
    b) UI mockups
    c) API, schema or format proposals
    d) Initial test plan – this may be quite skeletal, describing only testing methodology and key risk points which will need to be a focus of the implementation and testing
2) Phase: Implementation
    a) New software or service release
    b) User or developer documentation
    c) Release notes
    d) Final test plan
3) Phase: QA
    a) Test report
    b) Outstanding defect list

8. Systems Documentation

The Collaboratory hosts a documentation service. Platforms have been encouraged to deliver their public documentation as an HTML documentation bundle and as a living website. The documentation bundle is typically created from:

- Sphinx
- Doxygen

However, the Collaboratory documentation server supports any documentation system which generates HTML documentation.

9. User Support

Quality user support is a key aspect of the operation of any infrastructure, but this activity has only been lightly staffed in the Ramp-Up Phase and SGA1. However, Agile Methodologies provide a solution to this problem inside the boundaries of the current work plan. Scrum teams are made up of cross-functional feature development teams that actively engage user communities to understand their needs. Involving developers in the support process gives them key insights about usability enhancements and feature prioritisation needed for their software products.

With this in mind, HBP has adopted a practice of using software developers who are the primary support channel for the Platforms. This practice gives rise to Table 7, outlining support personnel breakdowns for the respective Platforms.

Table 7: HBP Support count, per SP

Support Role	COLL	SP5	SP6	SP7	SP8	SP9	SP10
Support Part-time — Developers	5	2	4	8	2	5	4
Support Part-time — Non-developers	0	1	0	0	0	0	2
Full-time	0	0	0	0	0	0	0
Total support (PT and FT):	5	3	4	8	2	5	6

These developers occupy support channels (see Figure 6) that will be found in various places in the Collaboratory. The Collaboratory has a global support request button for requesting email responses. This will be expanded to automatically file a support ticket in a pipelined user-ticketing system in SGA2 to allow users to track progress on their support requests and to allow easy handover to other Platform support teams. Finally, the Software catalogue links to the support channels for each of its entries. These support channels will either work to address the issue with existing features or add feature requests to the work Backlog for the Collaboratory, App or Software entry.

Wherever appropriate, the Collaboratory and App Support UIs will direct support requests to the HBP Forum[14] to allow the formation of a user knowledge base around frequently asked questions. It will also allow the community self-support.

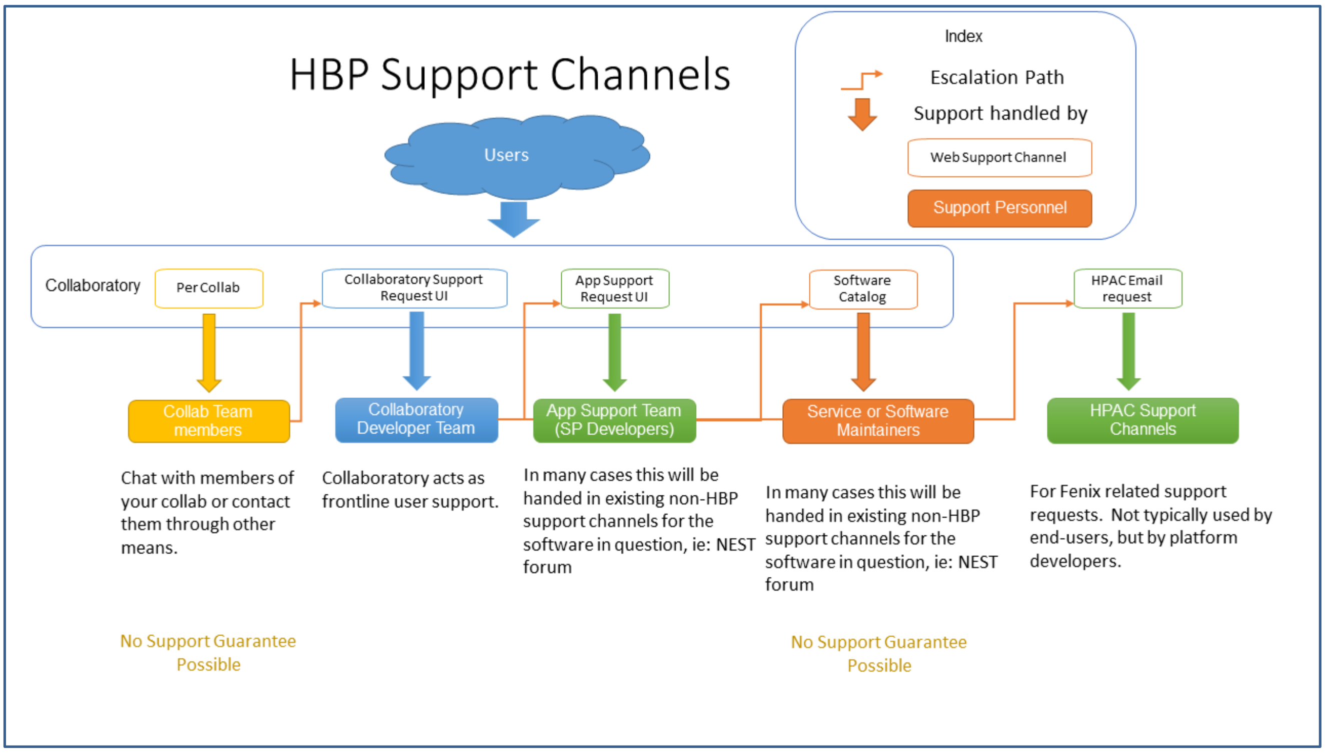

HBP Support Channels
Index
Escalation Path
Support handled by
Web Support Channel
Support Personnel
Users
Collaboratory
Per Collab
Collaboratory Support Request UI
App Support Request UI
Software Catalog
HPAC Email request
Collab Team members
Collaboratory Developer Team
App Support Team (SP Developers)
Service or Software Maintainers
HPAC Support Channels
Chat with members of your collab or contact them through other means.
Collaboratory acts as frontline user support.
In many cases this will be handed in existing non-HBP support channels for the software in question, ie: NEST forum
In many cases this will be handed in existing non-HBP support channels for the software in question, ie: NEST forum
For Fenix related support requests. Not typically used by end-users, but by platform developers.
No Support Guarantee Possible
No Support Guarantee Possible

Figure 6: HBP support channels

Annex A - HBP Extended TRL Standards

This annex describes the standards that must be applied by scientists, developers, and service operators in the HBP to apply a particular TRL label to a given system or service. The TRLs set out in Table 8 correspond to the standard European Commission TRLs.[15] They further expand the EC definitions and include the properties required of an infrastructure component at each TRL. The TRLs are intended to be applied to systems not only delivered as research infrastructure (RI), but also producing key datasets.

One caveat should be applied when applying Table 8: if systems are not continuously or near-continuously operating services, it should be assumed that where SLAs are specified they can be safely ignored in evaluating TRL criteria. SLAs would clearly not make sense in the context of a non-service system.

Table 8: Technology Readiness Levels

Technology Readiness Level	Expected Properties
TRL 1 Project Initiation	• Project owner identified • Project principles and high-level objectives defined • Use case definitions (includes target users and activities)
TRL 2 Conceptualisation	• Analytic study of the problem space • Identify key functions which must be validated in Component Implementation • Formulate validation criteria for critical components • Formulate validation criteria of complete prototype system • Prototype Epic planning
TRL 3 PoC Implementation	• Implementations of key functions • Validation of critical concepts • Identification of additional validation criteria for TRL4
TRL 4 Prototype Component	• Validation of prototype components in Lab • PoC has become prototype components • System technology selection has been made • Load testing of components under key load criteria • Identification of additional validation criteria for TRL5
TRL 5 Prototype Integration	• Validation of integrated system in a real-world environment • Tested in restricted environment with a small number of real users • Data formats specified • Identification of additional validation criteria for TRL6
TRL 6 Prototype-to-Real-world Integration	• Validation of integrated system in a real-world environment • Load testing of integrated system under expected load

	• Tested in a real-world environment with a small number of real users • Initial System documentation • Initial User documentation • System monitoring points specified (for services) • Identification of additional validation criteria for TRL7
TRL 7 Operational Integration	• Validation of integrated system in a real-world environment • Tested in a real-world environment with a small number of real users (canary testing for SoA) • System monitoring implemented (for services) • No expected data format or API changes (for services or software components) • Load testing of integrated system under expected load • SLA monitored (for services)
TRL 8 Deployment	• Validation of integrated system in a real-world environment • Tested in a real-world environment with a small number of real users • SLA enforced (for services)
TRL 9 Production	• Validation of integrated system in a real-world environment • Tested in a real-world environment with a target number of real users

Annex B — Infrastructure Tiers

The concept of infrastructure tiers in the HBP originated with the User Recruitment and Instructure Strategy Working Group (URIS-WG) whitepaper. However, this concept exists in many other federated infrastructure projects. In the HBP, services are assigned to tiers on the basis of the analysis of the PLA dependencies and an analysis thereafter by the Infrastructure Development Committee. The tiers are defined to highlight those areas where central quality control, extensive testing, SLAs and monitoring are necessary. It is also intended to highlight where addition flexibility is granted to the service provider.

The ***HBP Managed*** infrastructure tier will adhere to strict standards with centrally managed SLAs that guarantee high availability. A combination of essential Software and High-level Infrastructure that is federated over multiple sites will have to be committed, to achieve the necessary service availability. A support plan will be documented and will have resources committed. A sustainable roadmap for both Base Infrastructure and High-level Infrastructure forms the core of the HBP RI.

HBP Coordinated components of infrastructure are provided for and owned by individual partners (partner institutions and conglomerates or SPs). Adherence to the HBP standards is optional and SLA will have negotiated availability. HBP Coordinated services will be deployed on a mix of HBP Managed and non-HBP Managed base infrastructure. All Apps and Services are monitored for health and availability by HBP Managed services. The respective partners manage support and provide the service prioritised to encourage adoption for their respective infrastructure components. The partners are responsible for assessing TRLs.

Community Coordinated software infrastructure is provided for and managed by a Third Party not involved in the HBP. Apps and services may be monitored, and the Third Party decides on SLA and Support levels.

Services delivered in the Ramp-Up Phase should be considered HBP Coordinated due to the TRLs, as well as a lack of standardised monitoring and federated deployment capacity. Services delivered in the Ramp-Up Phase will be evaluated for inclusion in HBP Managed Infrastructure services during SGA1.

Annex C - BBP Development and Deployment Standards

Co-funded by the European Union

BBP Standard Development and Deployment Process

Release 0.1

TBD

March 08, 2016

CONTENTS

1 Language Guidelines **1**
1.1 C++ 1
1.2 Javascript 5
1.3 Java/Scala 14
1.4 Python 18

2 Delivering and CI **29**
2.1 1. Submit your code to the Gerrit repository 29
2.2 2. Check the automated validation done in Jenkins. 29
2.3 3. Add reviewers 29
2.4 4. Code merge. 30

3 Further Resources **31**
3.1 Useful links 31
3.2 Languages 31

4 Monitoring infrastructure **33**
4.1 Goals 33
4.2 Monitoring data 33
4.3 Framework 34

5 Elastic Search **37**
5.1 Components 37
5.2 Cluster deployment and specifications 38

6 Graphite **41**
6.1 Goal 41
6.2 Components 41
6.3 Standalone deployment 41
6.4 Federated deployment 42
6.5 Deployment configuration 43
6.6 Relay rules 44
6.7 Performance measurement 45
6.8 Data retention policy 45
6.9 How to bring new data 46

7 Icinga **49**
7.1 Infrastructure 49
7.2 Creating your own probes 49
7.3 Icinga Event Stream 50
7.4 Event Collector 51
7.5 Availability API 52

8 Log Management **55**
8.1 How to index logs 55
8.2 How to visualize indexed logs 59

CHAPTER

ONE

LANGUAGE GUIDELINES

This section defines standards for the various major programming languages used in projects hosted on the infrastructure BBP provides to HBP.

1.1 C++

1.1.1 Coding Standards

- Follow the existing coding style of a project

This documents lists the common practices for the BBP C++ projects. It is meant as a companion to the Coding Standard, and is compiled from typical issues found working with legacy projects.

A code reviewer should reject the review if any of these rules is violated.

If the rationale for any of the rules below is not clear, ask the author who'll happily explain it. Feel free to propose additions.

1.1.2 CMake

- Each project needs to have the open source CMake tree integrated, as described in the README. It has to include 'Common' early in the top-level CMakeLists.txt, and should also use GitTargets and CommonCPack. The Hello project can be used for guidance.
- Common.cmake will set sensible C++ default warnings, and make all warnings errors. Fix the warnings, quite a few of them will be bugs in legacy software. Repeat for a release build. Use 'include_directories(SYSTEM ...)' to "fix" third-party code.
- Use sub projects for "internal" dependencies.
- Do not place generated files in the source directory. They have to go into CMAKE_BINARY_DIR!
- Do not use 'file(GLOB ...)' to compile source lists
- Use proper library naming (see CommonLibrary.cmake)
- Run and fix 'make cppcheck' (provided by Common)
- Put header and implementation files in the same top-level directory name '<project>/', using the same name for the final include directory, the namespace and project name

1.1.3 Common Coding Practices

These practices highlight commonly found anti-patterns, and are not a complete list:

- write the code assuming that it will be publicly accessible, e.g., to potential employers.

Hard rules

These are easily verifiable rules, for which I haven't had an exception to the rule yet.

- constness: Declare every variable and method const, unless there is a logical design reason against it, e.g., it's a setter method.
- No ifdef within a class declaration. This is asking for runtime ABI trouble.
- Always initialize all variables
- methods always start with a verb: getWidth(), not width().
- class declarations have at most one public, protected and private section, in this order
- no break in switch(): For the occasional case clause with no break or return at the end, insert an explicit '// no break;' comment to express the fact you didn't simply forget the break;
- no public member variables in a class. It's ok to have structs as long as they only hold data.
- use (s)size_t for any container counter or other counting variable
- no tabs in source code, use unix line endings
- no magic numbers: At least use one constant with a proper name.
- Declare local variable as late as possible and initialize them during declaration. Observe constness rule.
- no 'using namespace foo' anywhere. Selective 'using Foo::bar' is acceptable in cpp files. Using namespace aliases is acceptable in cpp files.
- Don't do double-negatives: 'if(condition) doFoo(); else doBar();' is easier to read than 'if(!condition) doBar(); else doFoo();'
- use container.empty(), not container.size() == 0
- avoid inlining of functions unless really necessary; is essential for not breaking ABI
- use the following strategy for object allocations:
- use members/stack variables with const reference parameter passing as much as possible
- well-defined ownership: each new has a corresponding delete, the owning object allocates and deallocates the object in symmetric places
- shared ownership: each new assigns to a shared_ptr, return a shared_ptr from the creator function and use the shared_ptr everywhere throughout the API
- Always compare floating point numbers using an epsilon, by preference using std::numeric_limits< >::epsilon()
- Use 'f' for float calculations. 'float i = k * 0.1 ' is 'float i = (float)((double)(k) * 0.1)', which is slower and might yield a different result then 'float i = k * 0.1f'

Soft rules

These rules may be violated if there is a conclusive argument. The reviewer has to understand and agree with the argument.

- header file order: from most specific/local to least specific/global
- includes in headers: reduce them to the max by using forward declarations (gathered in types.h) and the pimpl idiom
- early outs: Reduce indentation by using early outs wherever it makes sense.
- respect the 80 characters per line rule to improve readability

1.1.4 Discussion

Please formulate a question or add suggest rules below. They'll be answered or incorporated in a timely manner.

1.1.5 Coding Styles June 2 2015

- Spacing * space inside the parentheses in statements (if, for, switch, while, ...) * Scope for change: ideally file/class, but function minimum
- Braces * on a new line * Scope for change: ideally file/class, but function minimum
- Position of & and * * put close to the type, not the variable * Scope for change: file/class
- Comment style * C++ style, not C * Scope for change: file/class
- Function and class names * camel case, functions start with a verb and optionally a noun afterwards, class name are nouns * Scope for change: project
- Line length
  - 80 or 100
- new projects * MyCoolClass.h (file name same as class name) * folder naming lowercase folders * Scope for change: feasible wrt API changes, project wide
- leading underscore for private member variables & functions * Scope for change: file/class
- Pimpl * nested class Impl * unique_ptr (C++11) or raw pointer (C99) _impl; * contains all private (non-virtual) class members, and nothing else * try to keep all pimpl members public and to avoid pimpl class hierarchies * Scope for change: file/class
- Namespaces * directory name should match namespace name (nested namespaces / directories are reflected accordingly) * (no conclusion was reached regarding the use of subdirectories without a matching namespace)

1.1.6 Building

All C++ projects should use C++, CMake and Superproject. To set up a build, do: Before you start make sure you've got all the necessary build infrastructure available: Now get your software:

```
git clone ssh://bbpcode.epfl.ch/common/config.bbp
mkdir build
cd build
cmake -GNinja -DINSTALL_PACKAGES=1
```

If the build doesn't run, try to fix it or ask Stefan Eilemann or Daniel Nachbaur.

Meanwhile, start reading and understanding the **'SubProject<https://github.com/Eyescale/CMake/blob/maste**
concept of linking multiple projects into a single CMake build.

Now start coding (example with RTNeuron):

Note:

- Lunchbox does not compile with mpi wrappers on BG/Q (some potential conflict with DNS package.. -I/usr/include is missing...)
- It compiles fine with XLC compilers
- Clone the Hello Project for new projects

1.1.7 Testing

- Use CTest, recommended boost::unit_test

1.1.8 Packaging

- See CPack setup in Hello Project

1.1.9 Profiling

- VTune

1.1.10 Delivering and CI

1.1.11 Debugging

TotalView : https://bbpteam.epfl.ch/project/spaces/display/BBPHPC/Software+Debugging

1.1.12 Documentation

1.1.13 Environment Setup

1.1.14 Deployment best practices

Point of contact: Jean-Denis Courcol, Olivier Amblet

Warning

The standards define in this page are just a proposal, here to start a discussion.

1.2 Javascript

Javascript section is the main language discussed in this section, but it also addresses some of the structure and development of Web Components and Web UI as a whole.

1.2.1 Coding Standards

Based on AngularJS' code standards .

Linting

Automated verification of the Coding standards and general best practice can be done with jsHint. The configuration of jsHint is stored in the .jshintrc file at the root of the project. You can have a look at the documentation page to see the explanations for the settings.

A list of available plugins for different IDEs can be found here.

1.2.2 The standards (WIP)

Global Scope and Closure

Unless necessary try not to create a new namespace instead write your code in an anonymous function like this:

```
(function(){
  /* your code goes here*/
}());
```

This will make sure you're not polluting the global scope.

Equality

Use the identity operators (=== and !==) to the equality operators (== and !=). If some type coercion is needed do it explicitly.

AngularJS specifics

https://github.com/mgechev/angularjs-style-guide

Building

Note

The build system we use is highly configurable and some of the explanation bellow may only be valid for the way our project is configured.

Overview

Here we follow the 2014-ish standard for creating client side web application (see

Choosing a way to managing Javascript artefacts). The development tool and build system (and perhaps a javascript server side app) are managed by the Node Package Manager(NPM). The client side dependencies (aka the components and the client javascript libraries like Angular) are managed by bower.

1.2.3 Installation

The tools we use to build frontend code are all based around Node.js and NPM. This means that you will need to install them. Sadly the current version in Ubuntu's repository are way to old. This means that you will have to install them yourself. This pages explain how to do this without sudo access: https://gist.github.com/isaacs/579814 (you can choose any of the proposed solution but I only tested the first one).

Once those two are installed you will have to install yeoman and bower with the following command:

npm install -g yo npm install -g bower

1.2.4 Bower

Bower is the package dependency manager that allows us to retrieve and resolve the dependency of our application. To download all the dependency automatically just execute the following command from your project root folder:

```
bower install
```

To add or remove a dependency to your project you will need to edit the file **bower.json** . It's structure is pretty obvious and you can look at

bower's documentation

to learn more. If you added a new dependency just run bower install

again to download it. If you removed dependency from the config file and want to remove them from your file system you will have to call

bower uninstall package_name_goes_here explicitly.

If you know a new version of a dependency exists, you can update your packages with bower update

Warning

It's very easy to add a package to bower's centralized repository. To make sure you don't inadvertently do this with one of our package you absolutely have to always specify

"private": true

in your **bower.json** file.

1.2.5 Grunt

We use Grunt to build the front-end code. It can be used for several purpose:

1.2.6 Development

If you use the command:

```
grunt server
```

It will start a server and open the app into the default website. When you modify a file it will automatically be deployed to the website, furthermore if you work with files that need to be compiled (like sass, compass or coffee) it will be compiled too. We've also configured the task so that javascript unit test are ran each time you same a js file.

If you want to test the production code in real time you can write this command instead:

```
grunt server:dist
```

If you want to have the auto-compile and error testing feature without starting the server you could write:

```
grunt karma:watch watch
```

1.2.7

1.2.8 Building

Grunt can also be used to generate the final application. This is done simply by calling:

```
grunt build
```

The resulting files will be stored in the dist folder.

1.2.9 Unit Testing

The unit tests are executed during the build and while the development server is running but if you want you can call it explicitly with:

```
grunt test
```

1.2.10 What to commit to your repository

The javascript community is divided on whether or not the dependencies and built artefacts should be checked in the final application version system. Statistically, the advantage goes to the groups that want to check everything in. But we think it is a bad habit that mostly come from the fact it is the easiest way to avoid a series of issues like:

- unable to deploy because an internet service is down (registry or repository),
- unable to reproduce a build because there is no locking(shrinkwrap) mechanism in bower.

A good rule of thumb is to avoid committing files that have been retrieved or generated by the build system. This includes all the downloaded modules and build files. I would recommend using a **.gitignore** file for this. Such a files is already define for the main project.

Ongoing trolling: Choosing a way to managing Javascript artefacts

1.2.11 Testing

Overview

We use grunt as a build system and to launch the tests too (for more information about grunt look at javascript build. Grunt uses Karma to run the tests. The tests themselves are written using Jasmin.

Configuration

The Karma configuration is in the **conf/karma.conf.js** file. The only reason you would want to modify this file is to add a dependency. This is done by adding an entry into the files array. The test files are automatically found if they are in the **test/unit/** folder.

Writing the Tests

The tests a written using Jasmin. They simply are js scripts situated into **test/unit**. They are ran inside PhantomJS. Since test are written in javacript it would be nice to stick to the javascript coding standards.

Running the tests

To run the test simply do:

```
grunt test
```

The result of the tests will displayed on the console as well as saved into a jUnit file (**unit.xml** in the root of your project).

1.2.12 Packaging

1.2.13 Profiling

Chrome developer tools : http://discover-devtools.codeschool.com/ Chrome Page Speed Plugin: https://developers.google.com/speed/pagespeed/

Web Page test : http://www.webpagetest.org/

1.2.14 Delivering and CI

Retrieving dependencies

Bower is used to retrieve client side dependencies, npm to retrieve server side dependencies (aka build dependencies). All the dependencies are made available through our internal registries that match a component name to a Git tag.

1. Dev ask JSBowerRegistry (using bower cmd) or Sinopia (using npm cmd) the location of a component.
2. The local registry fallback to the official public bower.herokuapp.com or registry.npmjs.org to get a result.
3. The result, containing a Git URL is retrieved to the Dev.
4. The component is retrieved at the desired semver by the command line tool from the Gerrit repository.
5. In the case of an external dependencies, the component is retrieved from there.

Interaction between a developer, Gerrit and Jenkins from the initial review to the release

(part in red are not implemented because of infrastructure issues)

1. When the developer post a review on Gerrit, Jenkins will be notified and a build will run *npm test* and *npm build* on the given patch set.
2. If all good, it will send a verified +1 notification to Gerrit
3. At some point, the user will merge its modifications to master. At this point jenkins will run *npm test* and *npm build*. In theory it should then commit the built artifact to Gerrit (bypassing the review).
4. When a developer want to make a release, he has to commit the wanted version through gerrit. Once all is in master, he connect to Jenkins and manually run a parametrised build where RELEASE_TYPE="RELEASE".

1.2.15 Project Structure

Here are the attended structure for the different type of projects:

Angular Module

An angular module is imported by one or more Angular applications.

Library Naming Convention Mutation (#scary)

Try to follow the following plan for the naming convention of an angular library:

Repository -> Component Name -> Main JS File Name -> Module Name ->

Declared entity name

JSLibAngularLibraryName -> angular-bbp-library-name -> ./angular-bbp-library-name.js -> bbpLibraryName -> libraryName

Corner cases:

- If the library is usable both by Angular or by a Vanila JS application, omit the angular in the naming chain.
- If the main file is different for Angular app and Vanilla app, use angular to prefix the main JS file that should be used by Angular

Minimal project structure

All those files must be checked-in.

```
/.bowerrc /.jshintrc /bower.json /Gruntfile.js /package.json /angular-bbp-library-name.js
```

The project structure might be fairly more complex but at the end, it should generate an angular-bbp-library-name.js file at the project root.

.bowerrc

bowerrc contains configuration relative to the bower command-line. The content of this file is exactly that:

```
{
  "registry":"'http://128.178.187.244:8080/bower"    <http://128.178.187.244:8080/bower>'_,
  "directory":"components"
}
```

bower.json

bower.json contains information about the library.

```
{
  "name":"angular-bbp-library-name",
  "version":"0.1.0",
  "authors": [
        "BBP/EPFL"
  ],
  "description":    "A    demo    component",
  "repository": {
        "type":"git",
        "url":"git+ssh://bbpcode.epfl.ch/platform/JSLibAngularLibraryName"
  },
  "main":"angular-bbp-library-name.js",
  "moduleType": [
        "angular"
  ],
  "private":true,
  "ignore": [
        "**/.*",
        "node_modules",
        "bower_components",
        "test",
        "package.json",
        "Gruntfile"
  ],
  "dependencies": {
        "angular":"~1.2.16"
  },
  "devDependencies": {}
}
```

Important:

- flag your package as private,
- indicate the repository.url using git+ssh:// protocol or no protocol at all (ssh:// is not supported by bower),

The dependencies should include only library that are not included into the built artefact, other should go in devDependencies.

It is a good practice to ignore everything excepted the files that needs to be included and the README. This way, the application does not weight more than necessary when checked into the repository with all its dependencies.

You can use *bower init* to generate a first version of this file.

package.json

package.json contains node informations for the component. Node is basically used to build the component using Grunt.

```
{
  ”name”:
“angular-bbp-library-name”,
  ”version”:
“0.0.0”,
  ”private”:
true,
”scripts”: {
   ”test”:
“grunt test”,
   ”build”:
“grunt build”,
   ”release”:
“grunt release”
  },
  ”author”:
“BBP/EPFL”,
  ”devDependencies”: {
   ”grunt”:
“~0.4.1”,
   ”load-grunt-tasks”:
“~0.1.0”,
   ”grunt-contrib-concat”:
“~0.3.0”,
   ”grunt-contrib-jshint”:
“~0.6.0”,
   ”jshint-junit-reporter”:
“~0.0.6”,
   ”grunt-contrib-clean”:
“~0.5.0”,
   ”grunt-release”:
“~0.7.0”,
   ”grunt-git”:
“~0.2.6”,
   ”request”:
“~2.34.0”
  }
}
```

Important:

- Jenkins plan will need the three specified scripts:
  - test : the command to run test
  - build : the command to create the distribution
  - release : the command to release the component
- Flag the component as private

Gruntfile

Gruntfile contains the build instruction.

```
/* jshint node: true, browser: false */
'use strict';
module.exports = function (grunt) {
  require('load-grunt-tasks')(grunt);
  require('time-grunt')(grunt);
  grunt.initConfig({
    jshint:          {
      options: {
        reporter:          require('jshint-junit-reporter'),
        reporterOutput:
'reports/jshint-unit.xml',
        jshintrc:
'.jshintrc',
      },
      all: ['**/*.js']
    },
    release:         {
      options: {
        file:
"bower.json",
        bump:
false,
// until jenkins can commit on master branch.
        commit:
false,
false,  push:
false   npm:
      }
    },

    gitcommit:     {
      dist: {
        options:   {
          message:
'built artefact',
          ignoreEmpty:
true
        },
        files: {
          src: mainjs
        },
      }
    }
  });
  grunt.registerTask('register',    function(){
    var request = require('request');
    var bowerConfig = grunt.file.readJSON('./bower.json');
    var  baseUrl  =  grunt.file.readJSON('./.bowerrc').registry;
    var done =
this.async();
    var  registerComponent  =  function(done)  {
      grunt.log.writeln('Send registration request');
```

```
(! (bowerConfig && bowerConfig.repository && bowerConfig.repository.url)) {
        grunt.log.error('Missing repository.url key in bower.json');
        done(false);
```

jshintrc

The .jshintrc file define the rules that should be enforced for the JS code format and quality. Developer's editor and Jenkins should use this file to test the lib.

You can download a pretty good version of this file from the JSLibOidcClient project, here .

1.2.16 Debugging

Chrome developer tools:

http://discover-devtools.codeschool.com/

1.2.17 Documentation

1.2.18 Environment Setup

Sublime Text

JSHint:

- install the package JSHint Gutter (shift+ctrl+P, then Package Control: Install Package, then choose JSHint Gutter).
- got to preferences / package settings / JSHint Gutter / Set Linting Prefererences.
- copy the content of .jshintrc of other bbp JS repositories
- execute linting by shift+crtl+j

Access Local NPM

Run the following command to be able to access and publish to bbpteam.epfl.ch/repository/npm (note that you must be on the EPFL network or connected via the proxy for it to work).

```
npm config set registry http://bbpteam.epfl.ch/repository/npm
npm config set email bbpsoatest@epfl.ch
npm config set _auth YWRtaW46bm9wYXNzd2Q=
npm config set always-auth
```

1.2.19 Deployment best practices

TBD

1.3 Java/Scala

Points of contact: Jeffrey Muller, Stefano Zaninetta, Yury Brukau.

1.3.1 Coding Standards

From here:

http://www.ambysoft.com/essays/javaCodingStandards.html See the attached javaCodingStandardsSummary.pdf

The following conventions on naming of methods and variables overrules the above document:

1.3.2 Naming

- Use short names for small scopes
  - is, js and ks are all but expected in loops.
- Use longer names for larger scopes
  - External APIs should have longer and explanatory names that confer meaning. Future.collect not Future.all.
- Use common abbreviations but eschew esoteric ones
  - Everyone knows ok, err or defn whereas sfri is not so common.
- Don't rebind names for different uses
  - Use final variables whereever possible
- Avoid using ‘s to overload reserved names.
  - typ instead of *type*
- Use active names for operations with side effects
  - user.activate() not user.setActive()
- Use descriptive names for methods that return values
  - src.isDefined not src.defined
- Don't prefix getters with get
  - As per the previous rule, it's redundant: site.count not site.getCount
- Don't repeat names that are already encapsulated in package or object name
  - Prefer:

```
object User {
   def get(id: Int): Option[User]
  }
  to
  object User {
   def getUser(id: Int): Option[User]
  }
  //They are redundant in use: User.getUser provides no more information than User.get
```

1.3.3 Documentation

Java code should be documented for JavaDoc: http://www.oracle.com/technetwork/java/javase/documentation 137868.html

The documentation should follow guidelines in the

JavaCodingStandards.

1.3.4 Building

1.3.5 Testing

1.3.6 Packaging

1.3.7 Profiling

Use Metrics

Official documentation of Metrics

http://metrics.codahale.com/manual/core/#man-core-meters

Setting a probe

Here I want to get the duration of the provenance query and the rate of its occurrence:

private final static

Timer responses = Metrics.newTimer(ProvenanceUtilities.class, "ProvenanceQueries", TimeUnit.MILLISECONDS, TimeUnit.SECONDS); public static OPMQueryResult getProvenanceHistory(EntityId karma) { final TimerContext context = responses.time(); OPMQueryResult s = runQuery(KarmaQueryPattern.sGET_PROVENANCE_HISTORY,karma.id()); context.stop(); return s; }

Launch your Jetty server.

Launch JConsole (Should be in the bin directory of your jdk)

Connect to your Jetty process through the "New Connection" submenu, Once

connected, go to the "MBeans tab".

important attributes are "Count" (the number of occurence) and "Mean" (average duration time of each occurrence).

Delivering and CI

Debugging

Environment Setup

Deployment best practices

Use maven to build.

Deploy through the BBP provided Nexus repository.

1.3.8 Scala Getting Started

Getting started

This is intended to guide you to scala resources that might be useful in your day to day work

Tools

Typesafe Debian Repository - Instructions on how to set up the Typesafe Debian repository is here:

http://typesafe.com/stack/download

- The Typesafe repo is used to install sbt below. Installation of the typesafe-stack package is optional * SBT aka Simple Build Tool - maven repositories and dependency management with a Scala DSL instead of XML. Lots of nice Scala friendly features -

  https://github.com/harrah/xsbt/wiki/Getting-Started-Setup
- Scala-IDE - a mature scala plugin for eclipse is here: http://scala-ide.org/download/current.html
- Recommended version of the Scala-IDE is currently 3.0
- Recommended version of Eclipse is 3.7
- Ensime - a scala ide for emacs users
- Giter8 - automatic Scala project template setup from github maintained project templates - https://github.com/n8han/giter8

Learning Scala resources

http://twitter.github.com/scala_school/

Documentation and Coding standards

- See here: https://bbpteam.epfl.ch/confluence/display/BBPWFA/WFA+Coding+Standards

1.3.9 Coding Standards

We follow this recommendation here. If the code deviates from the Style guide in an appreciable way, please file it as a bug or fix it. http://twitter.github.com/effectivescala/

ScalaDoc

Scala code should be documented for ScalaDoc: https://wiki.scala-lang.org/display/SW/Scaladoc

1.4 Python

Points of contact: Jean-Denis Courcol, Michael Gevaert, Juan Pablo Palacios.

1.4.1 Coding Standards

Refer to the BBP Python Best Practices document, attached

1.4.2 Repository Structure

Project Organisation

Refer to Python Package and Repo Naming

In each module, there are init.py and version.py files. The version code is described under 'Versioning':

_init .py:

```
""" the bbp provenance server """
from provenance_server.version
import
VERSION as   version    # pylint: disable=W0611
```

version.py:

```
""" provenance server version """

VERSION =  "0.0.2.dev4"
```

The VERSION must be in this exact format: uppercase, and using double-quotes

With this in place, one can simply import the version string in the setup.py, and only ever have to update it in the version.py file:

setup.py:

```
from provenance_server

import   version

...
```

```
version=  version
```

Setting up a new python repository

This paragraph describes how to set up a new python code repository following this project organization and naming conventions .

install bbp-platform-dev package

this script snippet requires to have python-virtualenv package installed.

```
# create a directory where you want to install the package

mkdir ~/dev-tools
cd ~/dev-tools
# create a virtual environment in that directory
virtualenv venv
# activate it (this should change your shell prompt) source
venv/bin/activate
# install an older pip version that is compatible with our build tools pip
install pip==1.5.4
# install the package

pip install -i http://bbpgb019.epfl.ch:9090/simple bbp-platform-dev -pre
```

generate the repository structure

```
# letGs clone your empty gerrit repository (hbp-my-repo):

cd ~/mystuff

git clone ssh://bbpcode.epfl.ch/platform/hbp-my-repo
#    in    the    virtualenv    you    just    created
generate_repository.py ~/mystuff/hbp-my-repo
```

Now your repository is ready. You just need to fill the readme.txt, git add everything, commit and review.

1.4.3 Installation of Packages

The list of Python software maintained in by the platform team can be found

here .

Setup

In general, one should always use a virtual environment to install packages, along with pip. Make sure that python-virtualenv is install on your machine:

```
#in Ubuntu/Debian

sudo apt-get install python-virtualenv

#in RedHat

$ sudo yum install python-virtualenv

#Next, create a virtual environment:

$ virtualenv venv

#activate it

$ source venv/bin/activate

#install the package you want

$ pip install -i http://bbpgb019.epfl.ch:9090/simple/ task-sdk
```

1.4.4 Building

The majority of building happens in jenkins. To test whether your packages are being built properly, you can:

```
make pypi-sdist
```

This will create and populate a 'dist/' directory.

1.4.5 Testing

Guidelines

After following the #Project Organization, the following holds true for testing:

1. Unit tests just test the sub_module.
2. Cross module just test multiple sub_module inside the same module. There are actually unit tests too.
3. Integration module tests require resources that are not provided by the repository (like access to a database for instance).

4. functional tests works on actual data, or take longer because of performances tests, or end user like tests.

#1,#2 should not take more than ~10s per test, and should, on average take < 1 sec #3 should not take in any case more than 5 min or are replayed by the continuous integration at each merge and for each automated review. #4 can take more time and are replay periodically so that they don't block the build. #4 are not taken into account in the coverage to avoid "lazy" coverage (code is called but not really tested).

Functional Tests

Note: If one is organizing their tests in directories, don't forget to put 'test' somewhere in the directory, so that nose knows that tests live there:

Test Driven Development

nosy https://pypi.python.org/pypi/nosy is a good tool to help your TDD development. It will crawl your filesystem from a particular path and re-run specified tests when detecting modifications.

you just install nosy in your virtual environment, create a configuration file (setup.cfg is the default configuration file name) like this one:

```
[nosy]

# Paths to check for changed files; changes cause nose to be run base_path
= ./
glob_patterns     =     *.py
exclude_patterns = *_flymake.*
extra_paths = setup.cfg
# Command line options to pass to nose options
= -x
# Command line arguments to pass to nose; e.g. part of test suite to run

tests = tests/unit_tests.py
```

and run:

```
nosy
```

since setup.cfg is in the extra_paths, tests will be re-run if you change the configuration file.

1.4.6 Packaging

Managing setup.py

Normal setup.py options apply, the python docs .

To pass pylint, you may need to include:

An example setup.py is:

Unknown macro: 'toggle-cloak'.

Unknown macro: 'cloak'

1.4.7 Profiling

Algorithm

1. Start with builtin module
2. Use IPython (some tips)
3. Read some StackOverFlow
4. Ask Mike

1.4.8 Delivering and CI

Setup

Requirements

For the build and continuous integration systems to work, a number of requirements need to be fulfilled. These are general conventions that exist in the Python community, but ones that are required here. They include:

- All packages must use the setup.py package creation system, this comes with additional requirements, like having a README.txt file
- If there are dependencies, packages must include a requirements.txt file that lists their dependencies. All dependencies must be qualified by an exact version. Finally, a package that is to be 'released' (to be described later) must only depend on packages that do not contain any 'dev' suffixes (also to be described later)
- All packages must conform to the Project Organization filesystem layout described later in this document
- All packages must pass pylint and pep8 as well as tests to be suitable for inclusion in the PyPi server, even if they are development packages.
- All packages must have a version attribute at the top level of the package. This should be the same as the string used in the setup.py file. It is described in the 'Versioning' section of this file.
- All packages must be included in their top level (ie: git root) Makefile, ensuring that test coverage is correctly implemented. Read about how to set this up in the 'Makefile setup' portion of this document.

Versioning

The versioning convention that is required in for python packages closely follows PEP-0386. It is summarized as follows. The version number pseudo format is:

The dev suffix has special semantic meaning. Specifically, packages for with this suffix will be created and pushed to a pypi repository, as long as they pass linting and tests. Once the 'dev' is removed, only packages that go through the 'release' process (as described later) will be pushed to the PyPi server. The last N is reserved for patches, and shouldn't be included unless develop- ment has advanced, but older code needs small amounts of patching.

note that '1.0' > '1.0.devN'

Ignoring pylint and pep8 errors

"A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines." - Emerson

When it's necessary, pylint and pep8 errors can be ignored in the following manor:

- pep8: append '#nopep8' to the end of the line (note, spacing is also important
- pylint: append ' # pylint: disable=errno,errno,errno' to the end of the line

Please ignore errors sparingly, and document why if you're doing it at the global or local scopes (not per line)

Makefile setup

```
#modules that have tests
TEST_MODULES=rest_services/bbp_handlers/tests
provenance_services/bbp_provenance/tests tornado_swagger/tornado_swagger/tests
provenance_services/tests provenance_server/bbp_provenance_server/tests rest_server
#
#modules that are installable (ie: ones w/ setup.py)
INSTALL_MODULES=provenance_server provenance_services rest_server rest_services
tornado_swagger
#packages to cover
COVER_PACKAGES=bbp_provenance,bbp_provenance_server,bbp_circuit_utilities,bbp_rest_server
#documentation to build, separated by spaces
DOC_MODULES=tornado_swagger/doc
##### DO NOT MODIFY BELOW ####################
CI_REPO?=ssh://bbpcode.epfl.ch/platform/ContinuousIntegration.git
CI_DIR?=ContinuousIntegration
FETCH_CI      :=
    $(shell if
[ ! -d $(CI_DIR) ]; then
      git clone $(CI_REPO) $(CI_DIR) > /dev/null
;
    fi;
    echo $(CI_DIR) )
include $(FETCH_CI)/python/common_makefile
```

order for INSTALL_MODULES is important, if one package depends on another, they have to be ordered based on dependency.

Additional Options

*IGNORE_LINT: This variable can be used to ignore certain files and directories for the pep8, pylint and nosetest coverage directories. The format is very sensitive; it is one large regex used to filter files. EX:

```
IGNORE_LINT=bbp_client/bbp_client/oidc/oauth2client|bbp_client/bbp_client/oidc/apiclient|bbp_client/bbp_ workflow-items
```

Makefile targets

The most important ones:

Target	Description
devinstall	as install but in development mode to make in-place source changes (like 'pip -e .)
clean	clean everything generated by make
pypi-sdist	generate pip packages
pypi-clean	clean generated pip packages
test	run the unit and integration tests
functional_test_all	run all the functional tests.
functional_test run	cross-package functional tests. There are also functional_test_<module> targets for individual packages.
verify_changes	run pep8, pylint, unit and integration tests, and coverage on (to be) commited changes
doc	generate documentation
doc-clean	clean generated documentation
help	this help

1.4.9 Release Process

Goal

The goal of the release process is reliably create packaged code that can be installed through a deployment system like pip. In order to ensure high quality of released code, this process is automated as much as possible. This should mean that the created packages have a high degree of uniformity.

Algorithm

Proposal

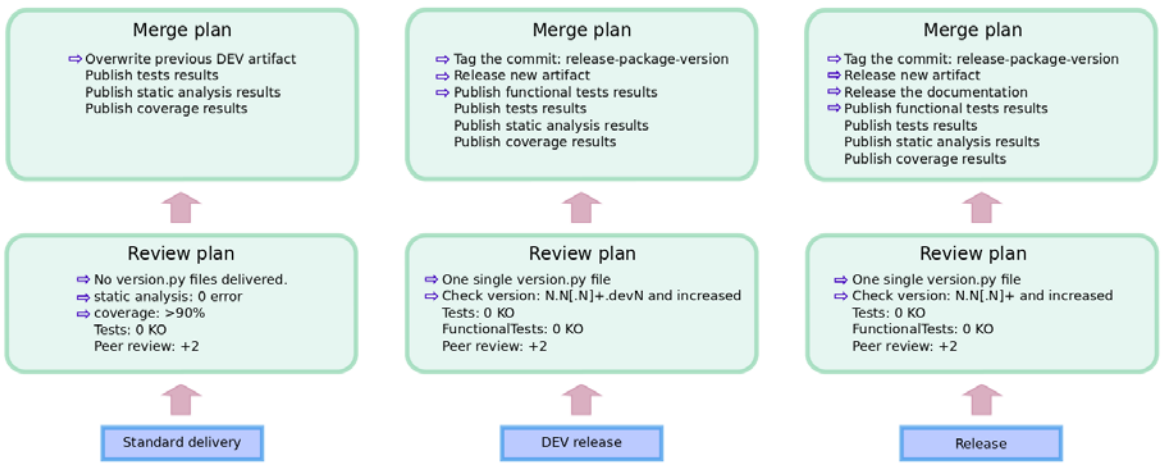
When a developer has changed the version.py file and updated the VERSION string, they would then do a 'git review'. This would kick off the the following algorithm:

In this changeset, the only file that can have changed is the version file.

1. Check if other files have changed in this change set, fail if so
2. Check if there is already a tag'd version with our target tag name (based on the package version.py), fail if so
3. Check that the version number is an increase on the previous one, fail if it's not. Remember to handle patches correctly.
4. Build the documentation with that name, and store it in a folder, and include it in the commit so that it's forever available, and push this to a documentation server
5. Tag the git commit with a tag name like what is used in the requirements.txt file, ex: 'bbp-provdmservices==0.0.2dev5'

Once the review happens, and the tag is accepted, there is a jenkins plan that uses the 'SSH Agent' option that has a key to push to the pypi repository, with the proper name, as created above

The dev suffix has special semantic meaning, refer to the 'Versioning' section.


Merge plan
⇨ Overwrite previous DEV artifact
Publish tests results
Publish static analysis results
Publish coverage results
Merge plan
⇨ Tag the commit: release-package-version
⇨ Release new artifact
⇨ Publish functional tests results
Publish tests results
Publish static analysis results
Publish coverage results
Merge plan
⇨ Tag the commit: release-package-version
⇨ Release new artifact
⇨ Release the documentation
⇨ Publish functional tests results
Publish tests results
Publish static analysis results
Publish coverage results
Review plan
⇨ No version.py files delivered.
⇨ static analysis: 0 error
⇨ coverage: >90%
Tests: 0 KO
Peer review: +2
Review plan
⇨ One single version.py file
⇨ Check version: N.N[.N]+.devN and increased
Tests: 0 KO
FunctionalTests: 0 KO
Peer review: +2
Review plan
⇨ One single version.py file
⇨ Check version: N.N[.N]+ and increased
Tests: 0 KO
FunctionalTests: 0 KO
Peer review: +2
Standard delivery
DEV release
Release

1.4.10 Debugging

1.4.11 Documentation

Project documentation can be quickly and easily generated by using sphinx . Most code that is to be accessed from outside of the platform team should have complete documentation of the accessible APIs.

Use of Sphinx

Installation

Installation of sphinx is accomplished with pip:

(The sphinx-napoleon extension is used to autogenerate API documentation, more on this later)

Sphinx Documentation Layout

The raw documentation resides in a directory called 'doc' which is at the same level as the setup.py or requirements file. For instance, for the tornado_swagger package, the directory structure is this:

Setting Up Sphinx

Once the doc directory is created, and after you have sphinx installed, do the following:

And follow this guide on how to configure it:

- Separate source and build directories - yes
- autodoc - yes (this can be added later)
- viewcode - yes (this can be added later)
- Create Windows command file? - no

An example run is:

Unknown macro: 'toggle-cloak' .

Unknown macro: 'cloak'

Once this process is completed, one can look at the file generated in the 'source/conf.py' directory. The extensions agreed to above (autodoc and viewcode) should be in the 'extensions' list. This file is normal python, and can import modules that are used in the autodoc, for instance.

It is recommended changing the theme to one that is much more usable:

To use the napoleon docstring formatting tool, add this to the 'extensions' list in the conf.py file: 'sphinxcontrib.napoleon'.

Using Sphinx

Now it's simply a matter of adding files to the 'source' directory, then adding that file to the 'source/index.rst' so that it is built and referenced.

To build the documentation simply:

```
% make html
```

A quick summary of some of the markup style is available here. A

cheat sheet is available here.

Note: Vertical whitespace is significant.

Having the version number autoupdated

By using the 'source/conf.py' mechanism, one can have the version number automatically in sync with what the package is.

For example, adding this to the conf.py:

along with the the version and release variables to existing module version:

```
version = tornado_swagger.  version

# The full version, including alpha/beta/rc tags.
```

should keep everything in sync.

1.4.12 Docstring standard

It was decided to use the Google standard for docstring documentation:

http://google-styleguide.googlecode.com/svn/trunk/pyguide.html?showone=Comments#Comments

This way, the 'napoleon' plugin can be used to extract relevant information and present it in the documentation. It is up to the developer to document or not parameter, types and exceptions, depending on the visibility of the API.

Rationale

- Pylint enforces in any case to provide a docstring.
- Raw 'sphinx' docstring are too dense to be used in the console
- 'numpy' style is nice but enforce type to be documented for return parameter and that is not pythonic
- Google style is a little bit less clear than 'numpy' but does not enforce type

1.4.13 autodoc examples

With the autodoc plugin enabled (either at project start time, or through the config file, documentation can be extracted from the docstrings. To do this, one creates declarations like:

The full documentation is here.

Once documentation is created and maintained, it should be periodically built and released along with the packages. This procedure has yet to be outlined.

The goal is to have the Python Continuous Integration system allow for full release control.

By having a release target, documentation will be generated, tagged and checked in, as well as pypi packages being generated and pushed to the proper repository.

- Use virtualenv.

CHAPTER TWO

DELIVERING AND CI

2.1 1. Submit your code to the Gerrit repository.

```
% git review
```

For a general overview of using Gerrit, go

here .

2.2 2. Check the automated validation done in Jenkins.

At this point, a Jenkins build will automatically run and performs the following checks:

- style and static code analysis of the modified file (ie: pep8 and pylint for python) The threshold is

  0 errors

  on

  modified files. Configuration files for static code analysis are located in the platform/ContinuousIntegration. You can submit a configuration change that will go through the code review process and will change the build configuration automatically once it is accepted.

- run the unit tests and the integration tests The threshold is

  0 errors

  on unit tests and integration tests.

- compute the code coverage The code coverage

  should not be below the previous value . The threshold is a monotonic function, that will increase until it reaches 90%. You can submit multiple patchset to fix fullfil the automated validation (git commit –amend, git review -R).

2.3 3. Add reviewers

Once the automated review is done, you can add reviewer so that they can focus on the latest patchset. Once a reviewer gives the +2, you can merge your changes through the "submit" button.

2.4 4. Code merge.

Once the code is merged, a build is automatically run that will :

- run the unit tests and integration tests (The functional tests may run on a separate and scheduled plan if they are too long).
- compute the code coverage, so that we have a consistent view of the evolution in Jenkins.
- execute style and static code analysis on the complete code base, so that we have a consistent view of the evolution for legacy code.

CHAPTER THREE

FURTHER RESOURCES

This section and subsections aims to contain all the information for Platform developers to be setup.

3.1 Useful links

JIRA:

- Issue Tracking for the platform: https://bbpteam.epfl.ch/project/issues/browse/LBK
- Platform Support queue: https://bbpteam.epfl.ch/project/issues/browse/PHELP
- Infra service desk: https://bbpteam.epfl.ch/project/issues/servicedesk/customer/portal/3
- Continuous Integration for the platform: Jenkins - https://bbpcode.epfl.ch/ci/view/platform/
- Code review: Gerrit UI - https://bbpcode.epfl.ch/code
- Python repository: Devpi - http://bbpgb019.epfl.ch:9090/
- Java and Web artifact repository: Nexus - https://bbpteam.epfl.ch/repository/nexus/index.html#welcome
- Package repository: http://bbpteam.epfl.ch/repository/ubuntu/
- Platform sites/services - see Platform operations
- Gitolite - to be used for personal repos only - Version control service
- VM management: Foreman - https://bbpcfmgr.epfl.ch/users/login
- BBP Openstack: https://bbpopenstack.epfl.ch/
- Source code search and cross reference: Opengrok - https://bbpcode.epfl.ch/source/
- Slack: https://collaboratorytest.slack.com
- Asana : http://www.asana.com

3.2 Languages

- Python
- C++
- Javascript
- Java and Scala

CHAPTER

FOUR

MONITORING INFRASTRUCTURE

- Carlos Aguado, BBP Core Services
- Alexandre Beche, BBP Core Services
- Ben Morrice, BBP Core Services

IMPORTANT NOTE: The monitoring infrastructure described in the **Monitoring Infrastructure**, **Elastic Search**, **Graphite**, **Icinga**, **Log Management** sections is only available for services running on Blue Brain Infrastructure.

4.1 Goals

- Delivers a generic monitoring infrastructure to provide dashboards, reports, notification system and analytic tools.
- Collect log data, metrics from various sensors and status from check and store them persistently in a storage system (Elasticsearch proposed).
- Target all possible systems involved in the operation of the project: * Hardware fabric: servers/clusters, network, storage, workstations, printers, help nodes * Applications: management applications, end-user support, file systems, application servers
- Provide dashboard with different scope: * Sysadmin (high level of details) * Manager (overview)
- Provide reporting functionality
- Perform alerting based on recorded trends and complex analysis of series of events (OLAP/CEP)
- Provide analytic tools to drill in the log.
- Provide a way of correlating everything (timestamps based)
- Arbitrary long-term archival of performance indicators: HPC performance indicators, applications usage, efficiency of consumable resources

4.2 Monitoring data

There are 3 kinds of monitoring data which are collected:

- **Metrics** for performance data

   - timestamped key-value (“timestamp name value”)
   - stored in fixed-size database (rrd style) where resolution is lost over time (persistent storage at full resolution should be considered allowing other aggregation at full resolution for old data)
   - No security concerns

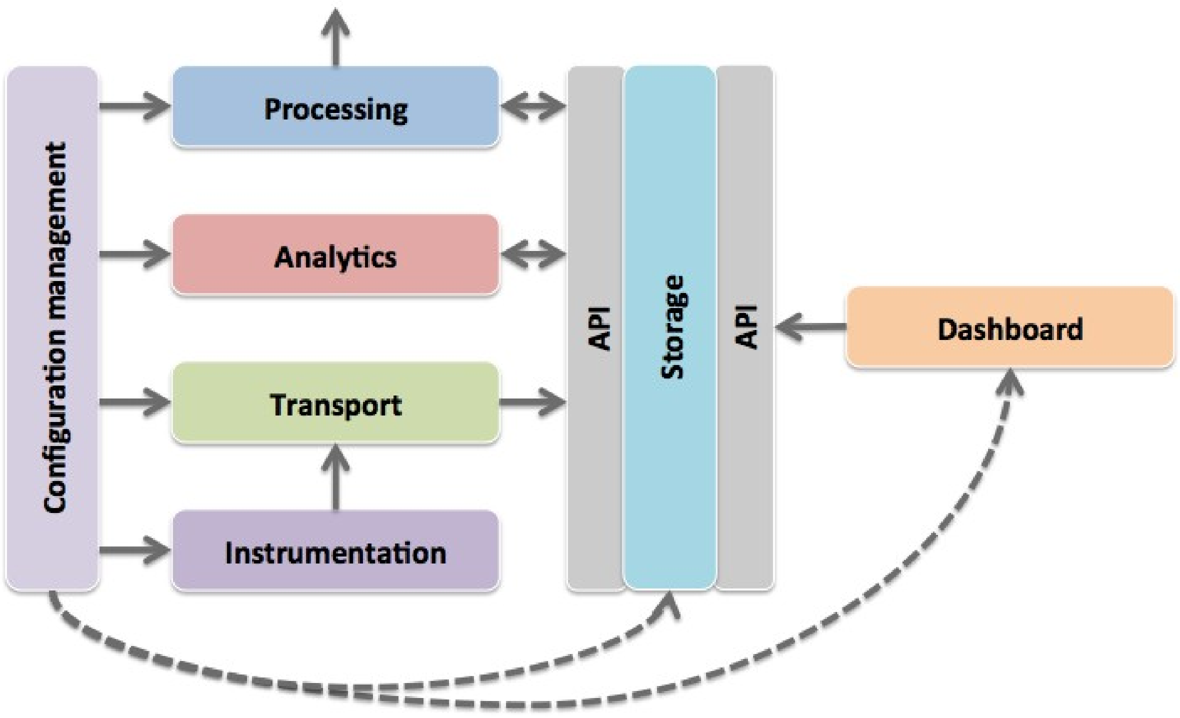
- **Status** for “nagios” checks * json entry made of service name, host name, metadata, time of status change and new status * Status are stored in nagios internals (can be retrieve by the livestatus plugin) * Status change should be stored (allowing service, host, ... availability) * No security concerns
- **Logs** * json entry made of metadata (host, cluster, application, ...) and free-text (log message) * Stored in Elasticsearch persistently (data lifetime?) * Security should be considered (how the logs are transported, who have access to them, which granularity of AuthZ)

In addition, an extra datatype is generated by the monitoring system:

- **Notifications** * event generated when some condition are met (triggered) * Can be email or RabbitMQ message

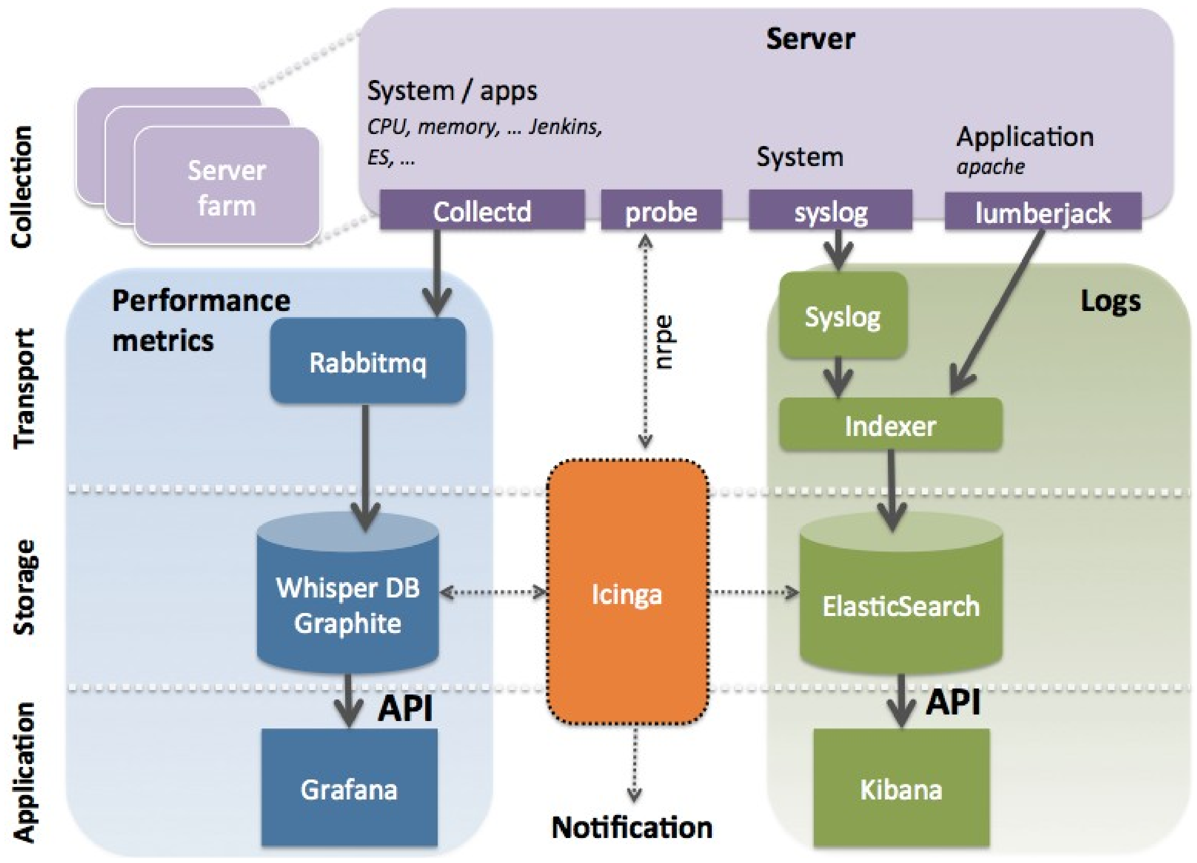
4.3 Framework

The monitoring infrastructure is organized in a well defined framework made of individual components communicating together through standard interface. The following framework has the advantage of being extensible and scalable since every component can be replaced with minimal impact to the other.



Implementation

- Monitoring configuration deployed through puppet


Server
System / apps
CPU, memory, … Jenkins, ES, …
System
Application
apache
Server farm
Collectd
probe
syslog
lumberjack
Collection
Transport
Storage
Application
Performance metrics
Rabbitmq
nrpe
Syslog
Logs
Indexer
Whisper DB Graphite
Icinga
ElasticSearch
API
Grafana
Notification
API
Kibana

CHAPTER

FIVE

ELASTIC SEARCH

Authors * Alexandre Beche, BBP Core Services

https://bbpteam.epfl.ch/project/spaces/display/INFRA/Elasticsearch

Goal

Elasticsearch is intended to index all the syslog messages of our infrastructure. In addition, it is now opened to store any kind of json documents.

5.1 Components

Elasticsearch is a distributed document store built on top of Apache Lucene. From

the configuration file, 3 interesting options can be set:

- **node.master:** <true|false> Is the current node eligible as a master
- **node.data:** <true|false> Is the current node able to locally store data
- **http.enabled:** <true|false> Is the current node able to serve http query

These options allow to define the flavor of an ES node.

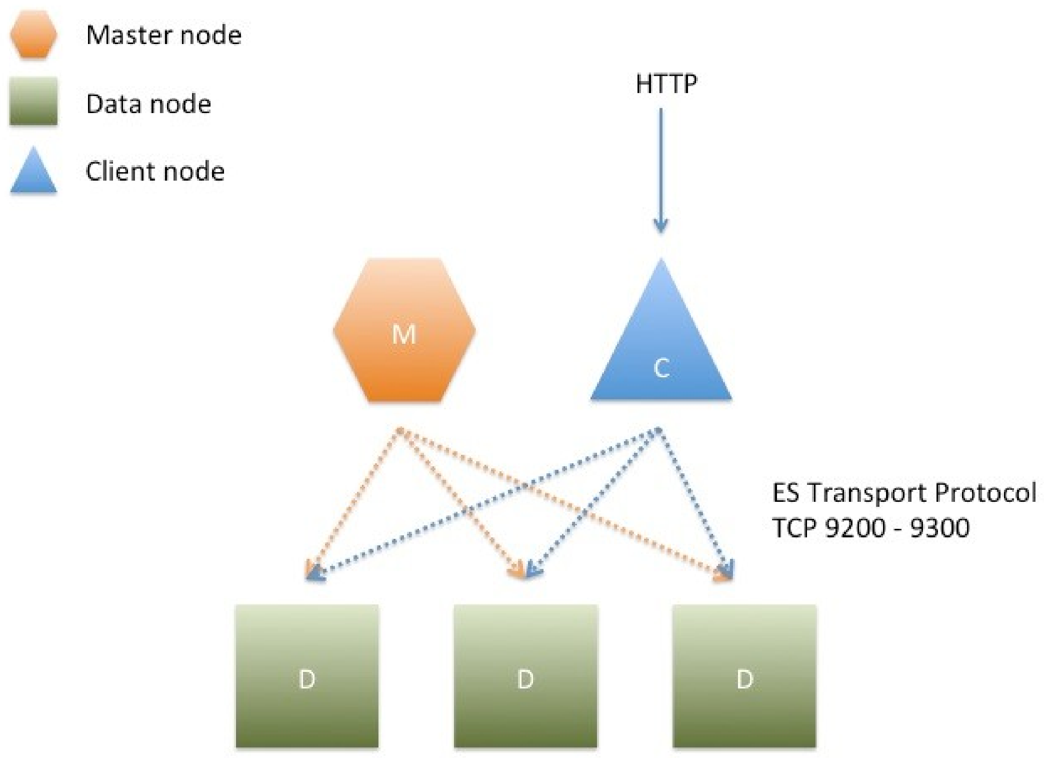
- **Master node:** Management node in the cluster responsible for shard allocation.
  - node.master=true, node.data=false, http.enabled=false
- **Data node:** Storage node responsible for storing the shards
  - node.master=false, node.data=true, http.enabled=false
- **Client node:** Proxy node to access the cluster through http
  - node.master=false, node.data=false, http.enabled=true
- **Standalone node:** A node able to play any roles (Good solution for testing cluster, shouldn't be used in production)
  - node.master=true, node.data=true, http.enabled=true

Visits 1 for more informations.

1 . http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/modules-node.html

5.2 Cluster deployment and specifications

The production cluster is currently running on 5 dedicated OpenStack nodes. Scalability can be ensure by scaling out data nodes or search node. Adding more master node would greatly improve the reliability by removing this single point of failure (however, getting 2 masters is not an option because of the split-brain problem).



While the master node is the most critical part of the cluster, it is the less resource-demanding, a small VM without any external storage is enough. The data nodes first require external storage(ie. 100G Ceph volume) and a lot of memory for indexing the data. Finally, the search node usually requires to run in-memory aggregations and sorting of the data returned by the data nodes.

	#	CPU	Memory	Storage
Master	1	2	4G	
Data	4	4	8G	100G
Search	1	4	8G	

Cluster optimization

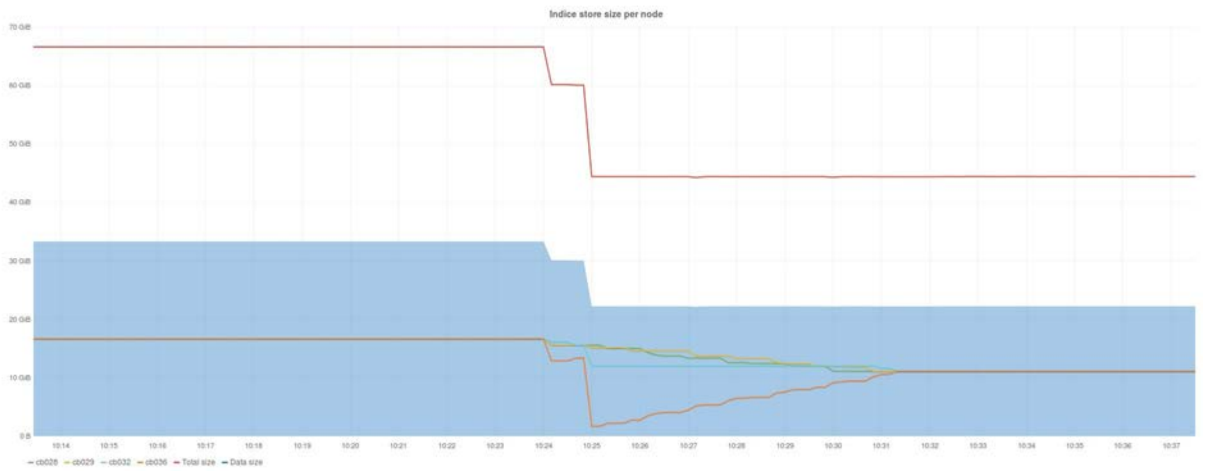
Number of shards and replicas is specific to any use-cases. The current configuration is 8 shards on 2 copies (1 primary + 1 replica)

Indexes backup and restore

Elasticsearch was designed with data locality in mind and shared storage should be avoided for "online" data. However some other use case can justified to backup the data (Full crash in a clus- ter, saving old data but removing them from the cluster).

Warning: Elasticsearch can snapshot/restore indexes which are closely tight to the Lucene index used. Better to backup the _source (ie. real data).

Cluster re-balancing after dropping a replica



CHAPTER

SIX

GRAPHITE

Chapter Authors * Carlos Aguado, BBP Core Services * Alexandre Beche, BBP Core Services

6.1 Goal

Graphite is intended to store timestamped data on the local file-system and query them through a HTTP API. Because this is a fixed size database, retention policy should be defined at the begin- ning and data get "compressed" (loose precision) over time. In the future we may think of storing persistently the whole history at higher precision in another data store such as OpenTSDB.

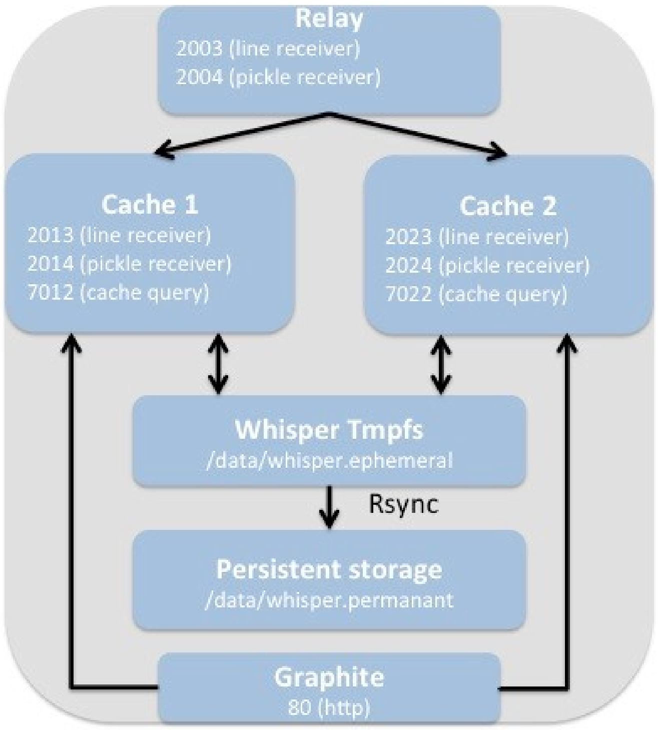
6.2 Components

- carbon-relay: "Router" that redirect metrics to another carbon-relay instance or to a carbon-cache.
- carbon-cache: Daemon that receives a stream of metrics (timestamped key-value pairs) and periodically flush them to whisper.
- whisper-db: Flat-file database format for storing time-series data. Each file represents 1 metrics and has a fixed size.
- graphite-web: Django application running the web API which query the carbon-cache.

6.3 Standalone deployment

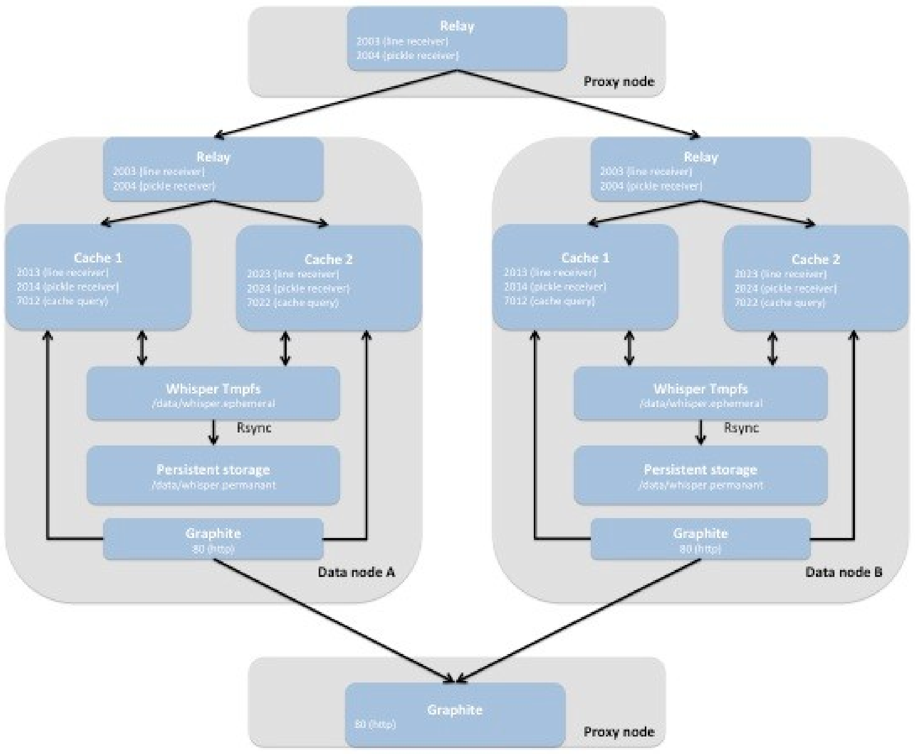
Because graphite is IO intensive, it has been decided to handle the whisper database in memory with a periodic rsync for the persistent storage (see performance improvements below).

On each graphite node, a relay is listening for incoming metrics and forward them to 2 carbon-cache process. Carbon cache is then flushing metrics to tmpfs, expose them through the query port to the graphite web api which serve them to the end user a JSON results (see diagram be- low).



6.4 Federated deployment

In order to scale graphite with the increasing number of metrics, a 5 nodes cluster has been setup. 1 node acting as a proxy for all incoming data and the federated view of the metrics tree, 4 data nodes for the storage setup as for the standalone deployment. (see figure below).



6.5 Deployment configuration

All the machine are puppet managed but a few configuration are still applied manually (mainly related to tmpfs configuration).

Puppet creates by default the folder for permanent sotrage (ie. /data/whisper.permanent)

6.5.1 creation of the ram disk

mkdir /data/whisper.ephemeral

mount -t tmpfs -o size=6g tmpfs /data/whisper.ephemeral/ ln

-s /data/whisper.ephemeral /data/whisper

6.5.2 Rsync script (including annotations)

vim /data/sync.sh

NOW=‘date “+%Y-%m-%dT%H:%M:%S” –date ‘now - 2 hours’‘

HOSTNAME=‘hostname |awk -F. ‘{print $1}’‘

curl -XPOST http://bbpsrvi47.epfl.ch:9200/annotations/annotate -d ‘

{

```
"@timestamp": "'$NOW"',
"title":  "rsync  on  '$HOSTNAME"',
"text":  "rsync  on  '$HOSTNAME"',
"tags":  "rsync-begin-'$HOSTNAME"'
}'
rsync  –archive  /data/whisper.ephemeral/  /data/whisper.permanent
NOW='date "+%Y-%m-%dT%H:%M:%S" –date 'now - 2 hours''
curl -XPOST http://bbpsrvi47.epfl.ch:9200/annotations/annotate -d '
{
"@timestamp": "'$NOW"',
"title": "rsync on '$HOSTNAME"',
"text": "rsync on '$HOSTNAME"',
"tags":  "rsync-end-'$HOSTNAME"'
}'
```

6.5.3 Crontab

```
crontab -e
1(/16/31/46) * * * * sh /data/sync.sh >/dev/null 2>&1
```

6.6 Relay rules

In order to be able to scale the cluster as the data store grows, it has been decided to route metrics based on rules. This method is supposed to easy the cluster re-balancing problem.

Configuration removed

6.6.1 Migration procedure

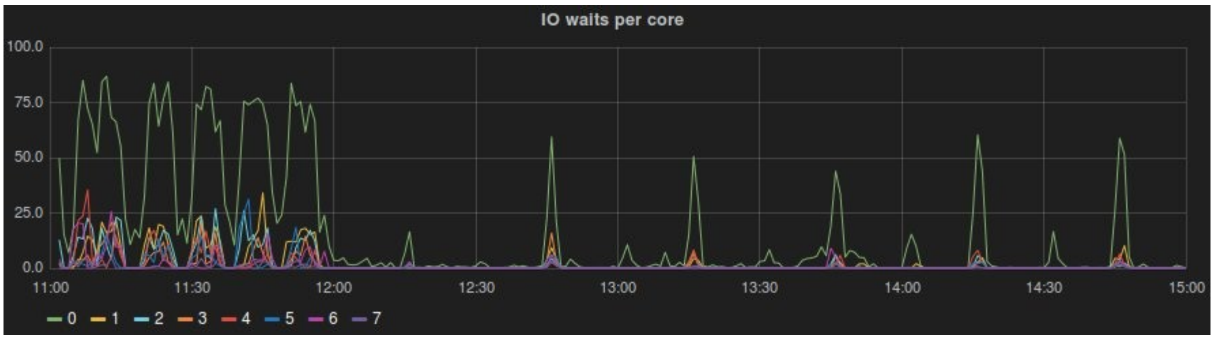
To move data across the nodes, the procedure should be the following:

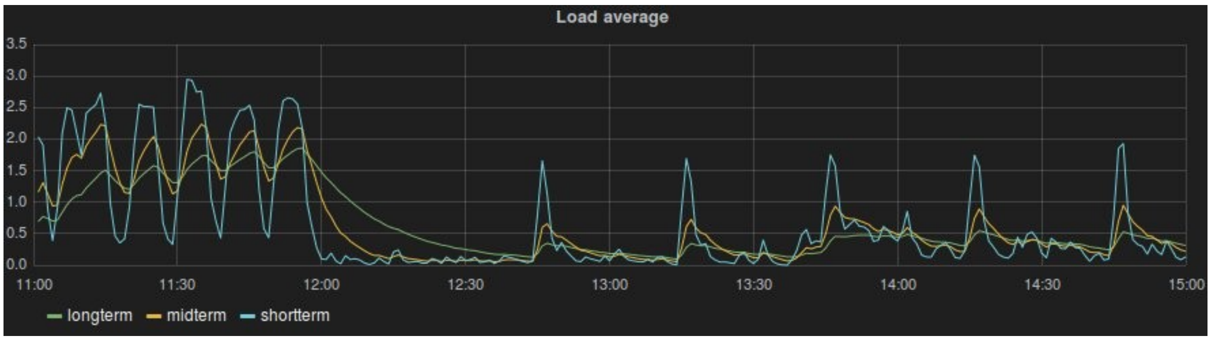
1. Disable puppet on the proxy node
2. Stop the collectd process which consume from rabbitmq
3. Stop the proxy relay (incoming metrics should stack into the broker)
4. Define the new rules into the proxy node (through puppet)
5. SCopy directory owning the metrics to the new host
6. Restart the relay, restart collectd and re-enable puppet (Backlog from RabbitMQ will be consumed)

No data loss should be observed

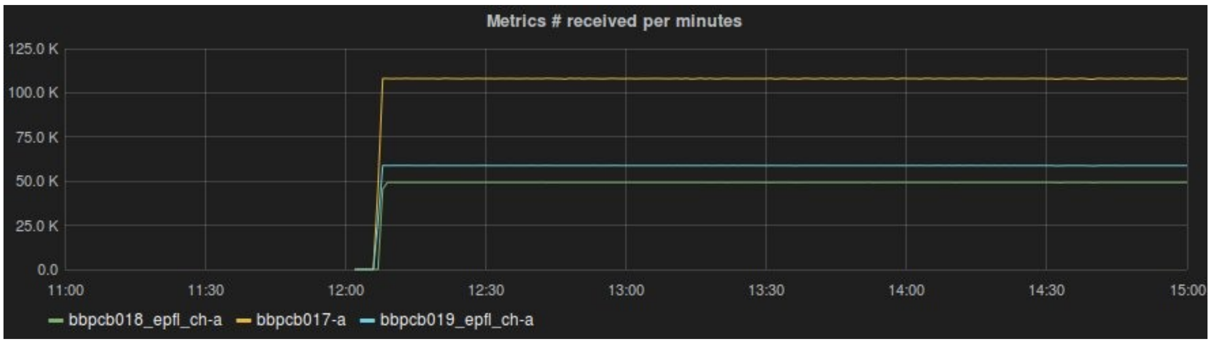
6.7 Performance measurement

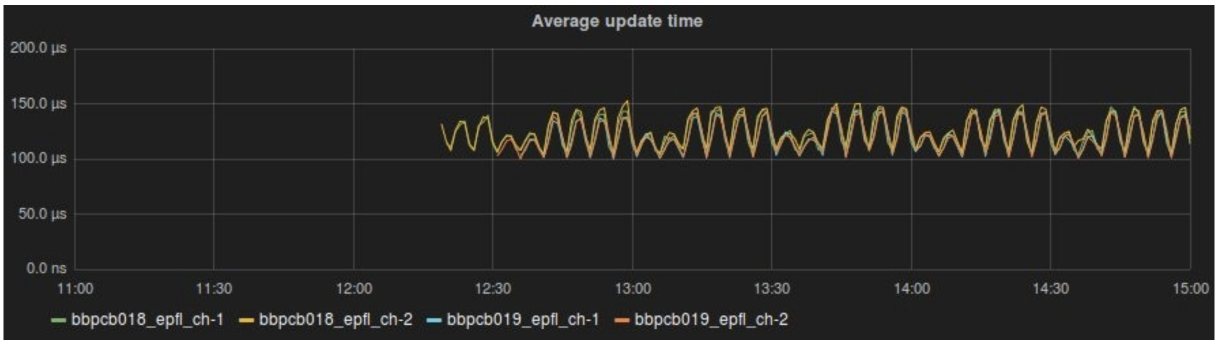
- At 12:00, whisper DB has been put in memory, we can see a good impact on the CPU IO waits and Load average of the machine.





- Currently our system is running without any problem with 110k metrics / minutes without any impact on the update time / metrics





6.8 Data retention policy

Since graphite is also used today as long term storage, we have a need of having:

- high resolution for fresh data
- reasonably long term history of aggregated data

The following default pattern has been implemented:

- 10s for 2 hours
- 60s for 2 days
- 5 minutes for 3 months
- 30 minutes for 6 months
- 1 hour for 2 years

This policy (can be overridden per-serie basis) represent 872 kB (852 kiB).

Given that we today have roughly 60k metrics, our storage capacity (and memory size) should be 48.8GiB.

6.9 How to bring new data

While new data can be added through the graphite UDP collector,this method is not recommendedand the associated port will be only opened for the RabbitMQ machine. Instead user should publish to the RabbitMQ server directly, this allow every monitoring probes to be storage-agnostic. Moreover, to ease the RabbitMQ interaction, it is highly recommended to use the Collectd daemon to send monitoring data (daemon being already installed on all the machine).

To decouple monitoring for streamline activity, we recommend not to instrument your application directly to send monitoring data but rather exporting its state through well-known API.

6.9.1 Write your own probes

Collectd offers a wide range of probes but give the ability of user to create its own. The recommended way of doing is by using the collectd-python binding (https://collectd.org/documentation/manpages/collectd-python.5.shtml). Below a skeleton of the python probes.

```
#import
the   collectd   library
import
collectd
# Define a variable which will be overridden by the configuration file
MY_VAR=
“default”
# Define the configure callback to read data from configuration file
def  configure_callback(conf):
  ”“”Received     configuration     information”“”
  global MY_VAR
  for
node  in  conf.children:
    if
node.key   ==
‘MyVar’:
      MY_VAR = node.values[0]
# Define a read callback which will send data through collectd
def read_callback():
 # Send a single value
 val = collectd.Values()
 val.host = <Service Name> #Workaround to not mess with system metrics val.type
 =
‘gauge’
val.type_instance  =
‘my-key’
 val.values = [my_value]
# Finally register
this
2
callbacks                    to                    collectd
collectd.register_config(configure_callback)
collectd.register_read(read_callback)
```

6.9.2 Configure Collectd to send data into rabbitmq

```
# Loads the Python plugin Plugin. Unlike most other LoadPlugin lines, this one
#should be a block containing the line “Globals true”. This will cause collectd
#to export the name of all objects in the Python interpreter for all plugins to
#see. If you don’t do this or your platform does not support it, the embedded
#interpreter will start anyway but you won’t be able to load certain Python
#modules, e.g. “time”.
<LoadPlugin      “python”>
Globals true

</LoadPlugin>
# Register
```

```
<Plugin “python”>
# Where your module sit on the FS
ModulePath “/path/to/module”

# Name of the python file
Import “collectd_plugin”

# Configuration of a given plugin
<Module “collectd_plugin”>
MyVar “whatever”

</Module>
</Plugin>
# Configure the amqp plugin to send data directly into RabbitMQ
LoadPlugin amqp

<Plugin “amqp”>
<Publish     “rabbitmq_publisher”>
Host “<rabbitmq_host>”

Port    “<rabbitmq_port>”
VHost   “<rabbitmq_vhost>”
User “<rabbitmq_user>”

Password   “<rabbitmq_password>”
Exchange    “<rabbitmq_exchange>”
RoutingKey “<rabbitmq_routingkey>”

</Publish>
</Plugin>
```

Then you can start the collectd daemon using the following command

```
# Test a read loop and exit
collectd -C my-collectd.conf -T

#Execute collectd in the foreground
collectd -C my-collectd.conf -f
```

CHAPTER

SEVEN

ICINGA

Chapter Authors * Alexandre Beche, BBP Core Services

Icinga (nagios killer) is a scalable and extensible monitoring system which checks the availability of your resources and notifies users of outages.

- Infrastructure
- Creating your own probes
- Icinga Event Stream
- Event Collector
- Availability API
  - Retrieving all the Status Changes for a service
  - Availability computation
  - Reliability computation

7.1 Infrastructure

	preprod	**devel / staging**
server	bbpmon02.epfl.ch	bbpmon03.epfl.ch
api	https://bbpmonitoring.epfl.ch/api	http://bbpmonitoring.epfl.ch/state
rabbitmq exchange	icinga.notification.preprod	icinga.notification.staging

7.2 Creating your own probes

Icinga probe must follow the nagios standard.

Plugin Overview

Scripts and executables must do two things in order to function as Nagios plugins:

- Exit with one of several possible return values
- Return at least one line of text output to STDOUT

The inner workings of your plugin are unimportant to Nagios. Your plugin could check the status of a TCP port, run a database query, check disk free space, or do whatever else it needs to check something. The details will depend on what needs to be checked - that's up to you.

Return Code

Plugin return code	Service state
0	OK
1	Warning
2	Critical
3	Unknown

Performance data

While nagios plugins API define a way of giving performance data, our implementation does not takes them into consideration. All performance metrics must be transfered through collectd as explained in

Graphite#Howtobringnewdata

Packaging & Deployment

BBP probes should be package as RPM and follow the naming convention nagios-plugins-bbp-*.

An example specfile can be found

ssh://bbpcode.epfl.ch/infra/monitoring.git

under

plugins/nagios/

The "client side" configuration of those probes should be puppet managed as in the following example:

```
# Installing monitoring probes for redis
package { 'nagios-plugins-bbp-redis':
ensure => present,
}
$cmd = '/usr/lib64/nagios/plugins/check_redis -w 128 -c 512 -t 1000'
$libdir = '/etc/nrpe.d'
icinga2::checkplugin { 'nrpe-check_redis.cfg':
checkplugin_file_distribution_method => 'inline',
checkplugin_libdir => $libdir,
checkplugin_source_inline => "command[check_redis]=${cmd}",
}
}
```

At this stage, your probes is ready to be checked by an Icinga server. To register in our central server, please contact core services.

7.3 Icinga Event Stream

The Icinga Event Stream (IES) has been developed to keep track of all the status changes in Icinga. To do so, a notification script has been hooked in icinga and report all event to a RabbitMQ exchange. Anyone is free to listen from this message queue and react on specific event.

Each event consist of a JSON payload containing the following entries: notificationtype, host, service, timestamp, state, description

{“description”: “DISK CRITICAL - free space: / 33669 MB (87% inode=98%); /dev/shm 3935 MB (100% inode=99%); /data 89334 MB (93% inode=99%); /data/whisper.ephemeral 24 MB (0% inode=97%);”, “service”: “disk”, “state”: “CRITICAL”, “timestamp”: “2015-04-08 09:40:38 +0200”, “notificationtype”: “CUSTOM”, “host”: “bbpcb020.epfl.ch”}

Connection details to the message queue:

host = bbpsrvi44.epfl.ch port = 5672 vhost = /icinga user = <ask-it> pass = <ask-it> exchange = icinganotification

7.4 Event Collector

The event collector has been designed to keep the service state history. It is designed in such a way that different data sources can be easily added and data-store (elasticsearch today) easily replaceable.

The logic is quite basic, the event collector is listening on the message queue for all the new events and add them to the statuschange index if 1) most recent know event for the couple host/service 2) status change since the latest record (removed duplicated).

The output field is populated only on problem (never when the service is recovering)

# Example of an event record in ElasticSearch

{

“_index”: “availability”,

“_type”: “statuschange”,

“_id”: “AUyUCm-ZJNnjUIBrrq0I”,

“_score”: null,

“_source”: {

“output”: “DISK CRITICAL - free space: / 33669 MB (87% inode=98%): /dev/shm 3935 MB (100% inode=99%): /data 89334 MB (93% inode=99%): /data/whisper.ephemeral 24 MB (0% inode=97%):”,

“host”: “bbpcb020.epfl.ch”,

“current_state”: 2,

“state_change”: “2015-04-07T15:20:47”,

“service”: “disk”}

}

Querying this history index should allow to recompute the history of a service.

7.5 Availability API

All status change are recorded into an ElasticSearch index and exposed through a well-defined HTTP API. The following opretations are available in the API.

7.5.1 Retrieving all the Status Changes for a service

Endpoint:

POST /api/getStatusChanges

Description:

This endpoint allow retrieving all the status changes (with/without including the initial state) for a given (hostname,service) tuple in a given time-window.

Parameters:

- hostname

: Hostname where the service is running

- service

: Service name for which to compute availability

- fromDate : Retrieve status changes from this date (default: now-1d)
- toDate

: Retrieve status changes to this date (default: now)

- initial

: Include the state just before the from-toDate interval

- order

: Order the data by timestamp asc / desc

Examples:

curl -k -H 'Content-Type: Application/json' -XPOST https://bbpmonitoring.epfl.ch/api/getStatusChanges

-d '{"hostname": "bbpmon02", "service": "icinga2"}'

{"data": [{"state_change": "2015-12-10T03:11:01", "foreman_environment": "preprod", "description": "PROCS OK: 1 process with command name icinga2", "service": "icinga2", "foreman_hostgroup": "monitoring/icinga2", "hostname": "bbpmon02", "current_state": 0, "fqdn": "bbpmon02.epfl.ch", "notificationtype": "RECOVERY"}], "params": {"toDate": "2015-12-19", "initial": true, "service": "icinga2", "hostname": "bbpmon02", "fromDate": "2015-12-14", "order": "asc"}}

7.5.2 Availability computation

Endpoint:

POST /api/getAvailability

Description:

This endpoint allow retrieving binned availability for a given (hostname,service) tuple in a given time-windowaswellasanoverallnumericalavailability(!basedondownsampleddata!)

Parameters:

- hostname

: Hostname where the service is running

- service

: Service name for which to compute availability

- services

: Bulk availability computation for hostname-service collection

- groups

: Allow grouping services together

- fromDate

: Retrieve status changes from this date (default: now-5d)

- toDate

: Retrieve status changes to this date (default: now)

- binSize

: Size of the bin to group the data (default: 1d)

- binFormat

: Format of the bin to be returned (timestamp by default)

- binOrder

: Ordering of the bins

Examples:

curl -k -H ‘Content-Type: Application/json’ -XPOST https://bbpmonitoring.epfl.ch/api/getAvailability -d ‘{“hostname”: “bbpmon02”, “service”: “icinga2”}’
{“data”: {“bbpmon02-icinga2”: [”ok”, “ok”, “warning”, “ok”, “ok”]}, “avails”: {“bbpmon02-icinga2”: 90}, “params”: {“toDate”: “2015-12-19”, “hostname”: “bbpmon02”, “fromDate”: “2015-12-14”, “service”: “icinga2”, “binSize”: “1d”}, “bins”: [1450393200, 1450306800, 1450220400, 1450134000, 1450047600]}
curl -k -H ‘Content-Type: Application/json’ -XPOST https://bbpmonitoring.epfl.ch/api/getAvailability -d ‘{“hostname”: “bbpmon02”, “service”: “icinga2”, “binFormat”: “%a. %d”}’
{“data”: {“bbpmon02-icinga2”: [”ok”, “ok”, “warning”, “ok”, “ok”]}, “avails”: {“bbpmon02-icinga2”: 90}, “params”: {“toDate”: “2015-12-19”, “hostname”: “bbpmon02”, “fromDate”: “2015-12-14”, “service”: “icinga2”, “binSize”: “1d”}, “bins”: [”Fri. 18”, “Thu. 17”, “Wed. 16”, “Tue. 15”, “Mon. 14”]}

7.5.3 Reliability computation

Endpoint:

POST /api/getMTMetrics

Description:

This endpoint allow retrieving the Mean Time to Recover (MTTR*), the Mean Time to Failure (MTTF*) and the Mean Time Between Failure (MTBF*) for a given service.

Parameters:

- hostname

: Hostname where the service is running

- service

: Service name for which to compute availability

- fromDate

: Retrieve status changes from this date (default: now-5d)

- toDate

: Retrieve status changes to this date (default: now)

- ignoreWarning : Warning are considered as a normal state if set (Critical Otherwise)

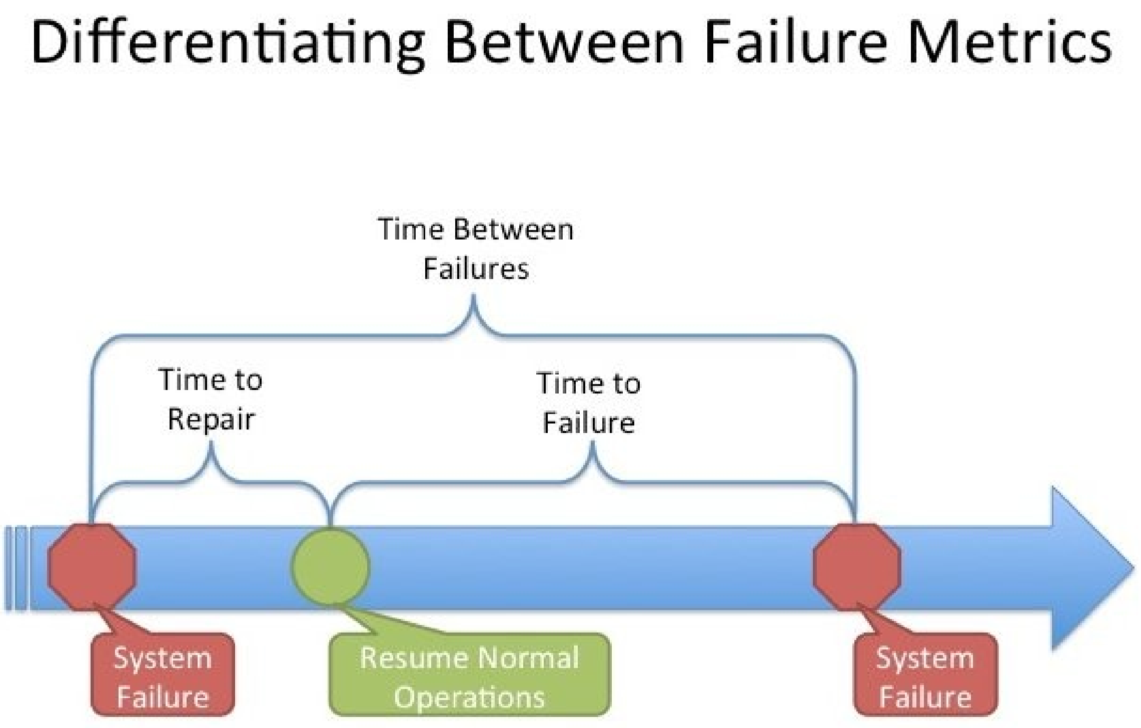
Examples:

curl -k -H 'Content-Type: Application/json' -XPOST https://bbpmonitoring.epfl.ch/api/getMTMetrics

-d '{"hostname": "bbpmon02", "service": "icinga2"}'

{"params": {"toDate": "2015-12-19", "initial": true, "service": "icinga2", "ignoreWarning": false, "hostname": "bbpmon02", "fromDate": "2015-12-14"}, "data": {"mttr": "7m 59s", "mtbf": "1h 12m 0s", "mttf": "3d 3h 38m 1s"}}

*Understanding reliability metrics:



CHAPTER

EIGHT

LOG MANAGEMENT

Authors * Alexandre Beche, BBP Core Services

8.1 How to index logs

Making the logs searchable from a central place is a 2 steps operation.

1. Shipping the logs
2. Indexing the logs

8.1.1 Shipping the log

Because log indexing is a cpu intensive operation which could potentially disturb the main activ- ity of a server, log file are not locally indexed but rather sent to a specific node dedicated to this task (ie. the indexer). In our infrastructure, log stash is used to perform this operation and con- figuration is 100% puppet-managed. To start shipping logs, the following puppet code should be added:

```
include ::env::logstash
env::logstash::input_file {
‘shipper-input’:
 path => [<path>], # required
 type => <type>, # required
 tags => [<list of tags>], # optional
}
env::logstash::output_redis {
‘shipper-output’:
 redis_key =>
‘<team-
name>’,
}
```

- Many input files may be given
- path in the input file support wildcards
- Type is an arbitrary string that define the type of your logs
- tags (optional) allow you to annotate your logs
- the following team name are supported: coreservices, hpc, neuroinformatics, neuro- robotics and platform

The following example show how one can ship nginx access logs and tagging them

```
include ::env::logstash
env::logstash::input_file   {
'shipper-input-nginx-error':
path            =>          [
'/var/log/nginx/access.log'
],
 type =>
'nginx',
 tags   =>   [$::environment,
'access']
}
env::logstash::output_redis  {
'shipper-output':
 redis_host          =>
'bbpcb023.epfl.ch',
redis_key            =>
'coreservices',
}
```

As a results, logs will reach the search engine (Elasticsearch) without being indexed.

```
{
 "message":
"128.178.97.66 - username [13/Aug/2015:10:42:26 +0200] "GET /tr HTTP/1.1" 401 194 "-"
"Mozilla/5.0
(X11;   Ubuntu;   Linux   x86_64;   rv:29.0)   Gecko/20100101
Firefox/29.0"",
 "@version":
"1",
 "@timestamp":
"2015-08-13T08:42:27.118Z",   !!!   Insertion   timestamp   by
default, not the log date !!!
 "type":
"nginx",
 "tags": [
"devel",
"access",
"coreservices"
],
 "host":
"bbpch015",
"path":
"/var/log/nginx/access.log"
}
```

8.1.2 Indexing the log

By default, logs are not indexed. That means free-text search is possible but answering question like "return all logs which were generated by anonymous user and return 404" is not.

Indexing is the operation which consist to interpret :

```
message =
“128.178.97.66 - username [13/Aug/2015:10:42:26 +0200] “GET /tr HTTP/1.1” 401 194 “-”
“Mozilla/5.0
(X11; Ubuntu; Linux x86_64;rv:29.0) Gecko/20100101
Firefox/29.0”“
```

into:

```
{
 ip:
128.178.97.66,
 user:              username,
timestamp:
13/Aug/2015:10:42:26
+0200
```

This operation requires a good knowledge of the syntax of the logs to be indexed. To do this operation Grok filters are used (https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html).

Grok pattern can be first tested in (http://grokconstructor.appspot.com)

Example:

```
#Line to be tested:
128.178.97.66
     · username [13/Aug/2015:10:42:26
+0200]
“GET /tr HTTP/1.1” 401
194 “-“
“Mozilla/5.0 (X11; Ubuntu; Linux x86_64;rv:29.0) Gecko/20100101 Firefox/29.0”
#Pattern to be applied:
^%{IPORHOST:clientip}      (?:-|%{USER:ident})      (?:-|%{USER:auth})      \[%{HTTP-
DATE:timestamp}\]          \”(?:%{WORD:verb}                %{NOTSPACE:request}(?:
HTTP/%{NUMBER:httpversion})?|-)”    %{NUMBER:response}     (?:-|%{NUMBER:bytes})
“%{NOTSPACE:referrer}” %{QS:UserAgent}
```

When the pattern match the log line, indexing operation can be tested as part of the log shipping workflow by adding:

```
env::logstash::grok_filter {
‘grok_filter_test’:
 type =>
‘nginx’,
 pattern=>
‘^%{IPORHOST:clientip}   (?:-|%{USER:ident})  (?:-|%{USER:auth})
\[%{HTTPDATE:timestamp}\]   \”(?:%{WORD:verb}    %{NOTSPACE:request}(?:
HTTP/%{NUMBER:httpversion})?|-)”    %{NUMBER:response}    (?:-|%{NUMBER:bytes})
“%{NOTSPACE:referrer}” %{QS:UserAgent}’,
}
```

Logs are by default indexed based on the insertion timestamp, not the “log date”. this behavior may be overwritten by knowing the field / format containing the log date. In the example below, we want to index the logs based on the "timestamp" field of the log line.

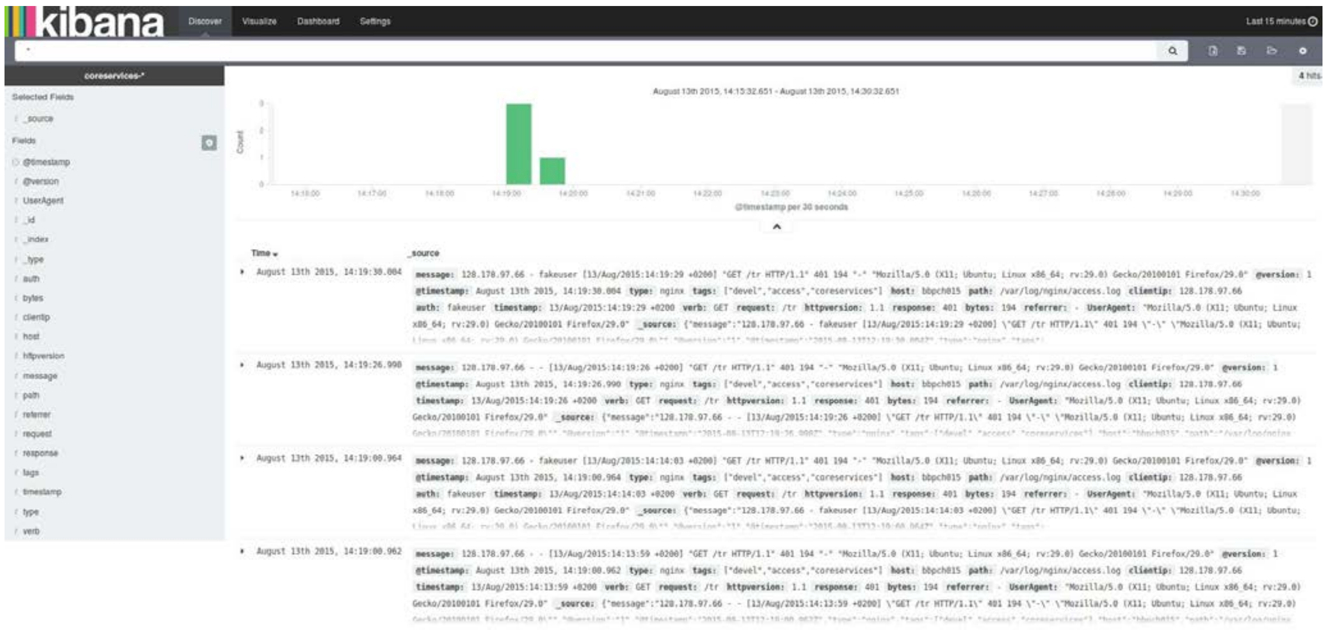
```
env::logstash::grok_filter {
'grok_filter_test':
  type     =>
'nginx',
  pattern =>
'^%{IPORHOST:clientip}  (?:-|%{USER:ident})  (?:-|%{USER:auth})
\[%{HTTPDATE:timestamp}\]   \"(?:%{WORD:verb}   %{NOTSPACE:request}(?:
HTTP/%{NUMBER:httpversion})?|-)"   %{NUMBER:response}   (?:-|%{NUMBER:bytes})
"%{NOTSPACE:referrer}" %{QS:UserAgent}',
  datefield      =>
'timestamp',
  dateformat                =>
'dd/MMM/YYYY:HH:mm:ss Z',
 }
```

If everything goes well, your logfile is now indexed as follow:

```
{
 ”message”:
“128.178.97.66 - fakeuser [13/Aug/2015:14:14:03 +0200] “GET /tr HTTP/1.1” 401 194 “-”
“Mozilla/5.0
(X11;   Ubuntu;   Linux   x86_64;   rv:29.0)   Gecko/20100101
Firefox/29.0””,
 ”@version”:
“1”,
 ”@timestamp”:
“2015-08-13T12:19:00.964Z”,
 ”type”:
“nginx”,
 ”tags”: [
“devel”,
“access”,
“coreservices”
],
 ”host”:
“bbpch015”,
”path”:
“/var/log/nginx/access.log”,
”clientip”:
“128.178.97.66”,
 ”auth”:
“fakeuser”,
”timestamp”:
“13/Aug/2015:14:14:03  +0200”,
 ”verb”:
“GET”,
 ”request”:
“/tr”,
”httpversion”:
“1.1”,
 ”response”:
“401”,
 ”bytes”:
“194”,
 ”referrer”:
“-”,
 ”UserAgent”:
“”Mozilla/5.0
(X11;   Ubuntu;   Linux   x86_64;   rv:29.0)   Gecko/20100101
Firefox/29.0”“
}
```

8.2 How to visualize indexed logs

Indexed logs can be queried / visualise from the test kibana interface. http://bbpcb023.epfl.ch:5601



```
August 13th 2015, 14:19:30.004  message: 128.178.97.66 - fakeuser [13/Aug/2015:14:19:29 +0200] "GET /tr HTTP/1.1" 401 194 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0) Gecko/20100101 Firefox/29.0" @version: 1
                                @timestamp: August 13th 2015, 14:19:30.004 type: nginx tags: ["devel","access","coreservices"] host: bbpch015 path: /var/log/nginx/access.log clientip: 128.178.97.66
                                auth: fakeuser timestamp: 13/Aug/2015:14:19:29 +0200 verb: GET request: /tr httpversion: 1.1 response: 401 bytes: 194 referrer: - UserAgent: "Mozilla/5.0 (X11; Ubuntu; Linux
                                x86_64; rv:29.0) Gecko/20100101 Firefox/29.0" _source: {"message":"128.178.97.66 - fakeuser [13/Aug/2015:14:19:29 +0200] \"GET /tr HTTP/1.1\" 401 194 \"-\" \"Mozilla/5.0 (X11; Ubuntu;

August 13th 2015, 14:19:26.990  message: 128.178.97.66 - - [13/Aug/2015:14:19:26 +0200] "GET /tr HTTP/1.1" 401 194 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0) Gecko/20100101 Firefox/29.0" @version: 1
                                @timestamp: August 13th 2015, 14:19:26.990 type: nginx tags: ["devel","access","coreservices"] host: bbpch015 path: /var/log/nginx/access.log clientip: 128.178.97.66
                                timestamp: 13/Aug/2015:14:19:26 +0200 verb: GET request: /tr httpversion: 1.1 response: 401 bytes: 194 referrer: - UserAgent: "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0)
                                Gecko/20100101 Firefox/29.0" _source: {"message":"128.178.97.66 - - [13/Aug/2015:14:19:26 +0200] \"GET /tr HTTP/1.1\" 401 194 \"-\" \"Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0)

August 13th 2015, 14:19:00.964  message: 128.178.97.66 - fakeuser [13/Aug/2015:14:14:03 +0200] "GET /tr HTTP/1.1" 401 194 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0) Gecko/20100101 Firefox/29.0" @version: 1
                                @timestamp: August 13th 2015, 14:19:00.964 type: nginx tags: ["devel","access","coreservices"] host: bbpch015 path: /var/log/nginx/access.log clientip: 128.178.97.66
                                auth: fakeuser timestamp: 13/Aug/2015:14:14:03 +0200 verb: GET request: /tr httpversion: 1.1 response: 401 bytes: 194 referrer: - UserAgent: "Mozilla/5.0 (X11; Ubuntu; Linux
                                x86_64; rv:29.0) Gecko/20100101 Firefox/29.0" _source: {"message":"128.178.97.66 - fakeuser [13/Aug/2015:14:14:03 +0200] \"GET /tr HTTP/1.1\" 401 194 \"-\" \"Mozilla/5.0 (X11; Ubuntu;

August 13th 2015, 14:19:00.962  message: 128.178.97.66 - - [13/Aug/2015:14:13:59 +0200] "GET /tr HTTP/1.1" 401 194 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0) Gecko/20100101 Firefox/29.0" @version: 1
                                @timestamp: August 13th 2015, 14:19:00.962 type: nginx tags: ["devel","access","coreservices"] host: bbpch015 path: /var/log/nginx/access.log clientip: 128.178.97.66
                                timestamp: 13/Aug/2015:14:13:59 +0200 verb: GET request: /tr httpversion: 1.1 response: 401 bytes: 194 referrer: - UserAgent: "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0)
                                Gecko/20100101 Firefox/29.0" _source: {"message":"128.178.97.66 - - [13/Aug/2015:14:13:59 +0200] \"GET /tr HTTP/1.1\" 401 194 \"-\" \"Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:29.0)
```

Annex D – BBP Python Development Standards

Blue Brain Project - Python Best Practices

Set of best practices to follow for current and future Python software within the Blue Brain Project.

Author: Valentin Hänel <valentin.haenel@epfl.ch>

Date: Apr 16, 2012

Version: 1.0

CheckoutURL: https://bbpteam.epfl.ch/svn/user/haenel/bbp-python-best-practices

SVN Revision : 3295

git hash : 3332bc71d67bee3612f2fbd20595380062ca24fb

Table of Contents

1 **Introduction** **3**
2 **The Ipython Interpreter** **3**
3 **Style Guide and Conventions** **4**
3.1 Style consistency 4
3.2 Code lay-out 5
3.3 Imports 6
3.4 Whitespace in Expressions and Statements 8
3.5 Naming Conventions 10
3.6 Exceptions 11
4 **Static Checking** **12**
4.1 Comparison 13
4.2 Example 13
5 **Testing** **14**
5.1 Unitest 15
5.2 Nosetest 16
5.3 Numpy Testing 16
6 **Documentation** **17**
6.1 Docstrings 17
6.2 Numpy Docstrings 18
6.3 File Header 19
6.4 Automatic API Generation 19
7 **Writing the `setup.py` file** **19**
8 **Versioning** **20**
9 **Obtaining Python Packages** **21**
9.1 Python on Linux 21
9.2 Python on Windows and Mac 22
10 **Wrapping Native code** **22**
11 **Performance Considerations** **23**
11.1 Define "Too Slow" 23
11.2 Identifying the Bottlenecks 23
11.3 Avoiding Premature Optimization 24
11.4 Consider time for Development 24
12 **Target Version** **24**
12.1 Python 2.x 24
12.2 Python 3.x 24

1 Introduction

This document collects the recommended best practices for programming in Python at the Blue Brain Project. This document contains both a coding standard, based largely on the existing, published style guide for Python code (PEP8 [3]), as well as additional recommendations for static checking, future proofing code, writing documentation, automated testing and packaging. The format of presentation is based loosely on the existing BlueBrain C/C++ coding standard.

This document aims to be succinct and self contained but also provides links to additional material for the interested readers where appropriate.

If you really have no time to read this document, you should at least look at "10 Ways to let People know you'r a bad Python Programmer [1]". Besides being quite informative, its actually also quite funny.

As a side note, the design choices and style guidelines for Python are conveniently summarised in 19 aphorisms which are accessible by typing `import this` in any interactive Python interpreter:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2 The Ipython Interpreter

The ipython [2] interpreter is the de-facto preferred interpreter for interactive work with Python. It has a plethora of extremely useful features which make working with Python interactively much more convenient. Since an in-depth description of Ipython is outside the scope of this document, we highlight three substantial productivity boosters.

- **History System** -- Akin to any UNIX shell, Ipython has a sophisticated history mechanism which can be used to recall previous commands.
- **Tab Completion** -- Again, like a UNIX shell, Ipython has a tab-completion system. This will complete Python keywords and built-in functions (e.g. `try`, `list` and `dict`), variables defined within the current scope and also executable system commands.
- **Easy Acces to Help** -- Using the question mark (?) after a function or method will display available help. This is usually much faster than using `dir()` since it is less to type and can be appended to the command line:

```
>>> list?

Type:       type
Base Class: <type 'type'>
String Form:    <type 'list'>
Namespace:  Python builtin
Docstring:
    list() -> new empty list
    list(iterable) -> new list initialized from iterable's items
```

For more information consult the website.

For more information on how to obtain Ipython, see the section on Obtaining Python Packages

3 Style Guide and Conventions

Code should be formatted according to the Python Enhancement Proposal #8: PEP8 [3]. It contains recommendations about:

- Style consistency
- Code lay-out
- imports
- Whitespace in Expressions and Statements
- Comments
- Documentation Strings
- Naming Conventions
- Programming Recommendations

Since this document is quite long, we summarize the most important aspects in the following sections and add some additional recommendations.

3.1 Style consistency

3.1.1 Prefer a consistent style

If you are modifying code that conforms to different recommendation than PEP8 [3] don't mix them. Reuse the existing style as much as possible.

For example, the standard Python naming conventions (see section on Naming Conventions) dictate that functions should use the style **lower_case_with_underscores**. However, due to the influence of Java and C++ many libraries use the style **mixedCaseFunctions**. In this case it is recommended to continue using this style, since mixing styles will be even more ugly than a non standard style:

```
class SpamHamAndEggs(object):

    def exportFooFromBar(self):
        pass


    # BAD PRACTICE: MIXING NAMING STYLE

    def export_bar_from_foo(self):
        pass
```

3.1 Code lay-out

3.1.1 *Never mix tabs and spaces and use four(4) spaces for indentation*

In python, whitespace matters. Mixing tabs and spaces will cause the interpreter to abort. According to PEP8 [3] you should use 4 spaces for each level of indentation.

```
for i in [1,2,3]:
    # first level indentation
    if i < 3:
        # second level indentation
        print "less than three"
```

Also It's a good idea to Configure your editor to adhere to the above. For example by automatically inserting 4 spaces instead of a tab character when you type the TAB key.

The exception here is if you are working with code, that either uses 8 spaces for each level of indentation or tabs. In this case, adhere to the existing style.

3.1.2 *Limit the line length to 79 characters and split lines logically*

Although many people argue that our monitors are large enough these days to support a line length of >79 characters, the PEP8 [3] strongly argues for the old 79 character limit which originates from the use of old UNIX terminals, which were indeed limited to 80 characters. The PEP8 [3] argues that default wrapping on such devices makes it more difficult to read. The argument that such devices are no longer prevalent, is invalid if one considers the recent advances of small form factor portables such as netbooks, ultrabooks and 12" laptops.

If you must split code across lines, use Python's implied line continuation inside parentheses, brackets and braces. Here is the example for the PEP8 [3]:

```
class Rectangle(Blob):

    def   init  (self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and

            color == 'red' and emphasis == 'strong' or
            highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or

                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                            (width, height))
```

Depending on your personal preference you could also use a backslash (\) to perform line continuation, preferably after the operator. The example above could thus be transformed as:

```
class Rectangle(Blob):

    def   init  (self, width, height,
                 color='black', emphasis=None, highlight=0):
```

```
        if width == 0 and \

                height  ==  0  and  \
                color == 'red' and \

                emphasis == 'strong' or \
                highlight > 100:

            raise ValueError("sorry, you lose")

        if width == 0 and height == 0 and \

                (color == 'red' or emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))

        Blob.  init  (self, width, height,
                      color, emphasis, highlight)
```

3.1.2 Use the blank lines to separate code

- Separate top-level function and class definitions with two blank lines.
- Method definitions inside a class are separated by a single blank line.
- Extra blank lines may be used (sparingly) to separate groups of related functions.

3.1.3 Prefer a portable she-bang

```
#!/usr/bin/env python         <-- CORRECT
#!/usr/bin/python             <-- WRONG
```

The rational is, that the Python interpreter may be in a location other than `/usr/bin` or there may even be multiple interpreters present on the system Using `env` (under the assumption that it is correctly configured) allows you to invoke the preferred interpreter for the system.

3.1.4 Prefer UTF-8 encoding

Here you can use the encoding magic:

```
# -*- coding: utf-8 -*-
```

This allows you to use correctly encoded UTF-* Characters, for example äöü or èéô in your source code, for example when documenting the names of the authors.

3.3 Imports

3.3.1 Place all imports at the top of the file

Unless you have a very good reason not too (for example if you have optional dependencies, see below). Group the imports in the following order system imports, third-party library imports and local application imports:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import  os
import sys
import numpy
import scipy


import bbp
```

One very good reason are optional dependencies. In this case it may sometimes be useful to place imports at the top of the function or method that requires them, for example:

```
def    optional_func():
    import optional_lib
    lib.compute()
```

This does not inpact performance, since imports are already cached. Meaning: in case the `optional_lib` has already been imported, it will not be imported again. This also means that you can't use `import` to re-import a module, instead use `reload`.

A second good reason is the import of functions from a module only within a function. In the following example the functions `cos` and `sin` are only used in in the function `cotan`:

```
def cotan(theta):
    from numpy import cos, sin
    return cos(theta)/sin(theta)
```

3.3.2 *Avoid relative imports*

Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports. Relative imports are hard to read and will easily break when refactoring. The following is a canonical example

Imagine you have the following package structure:

```
package
+--   init  .py
+-- sub1
    +--   init  .py
    +-- mod1.py
    +-- mod10.py
+-- sub2
    +--   init  .py
    +-- mod2.py
```

Where `mod1.py` contains the relative import, using the `..` notation:

```
from ..sub2 import mod2 as mod


print mod.func()
```

And `mod10.py` contains the absolute import:

```
from package.sub2 import mod2 as mod


print mod.func()
```

And `mod2.py` contains the code to be executed:

```
def func():
    return "func call"
```

Where both do actually work:

```
>>> import package.sub1.mod1
func call
>>> import package.sub1.mod10
func call
```

3.3.1 *Avoid using the star import*

```
from bbp import *
```

UNLESS you are working in the interactive interpreter. (c.f. **Namespaces are one honking great idea -- let's do more of those!**)

The rational is that importing everything from a name-space pollutes the name-space, may take a long time to load, may overwrite existing definitions and makes it hard to trace the origin of imported names.

There are two recommended ways to import:

```
from bbp import Neuron
n = Neuron()
```

Or alternatively:

```
import bbp
n = bbp.Neuron()
```

There is also a syntax using the *as* keyword:

```
import bbp as b
n = b.Neuron()
```

But for the BBP-SDK this is not so useful, since it has a short name already, and shortening it to a single character violates the rule about not naming variables with single letters (see also: the section on naming Variables). If the SDK had a different name, for example *blue_brain_project_sdk*, this would be more useful:

```
import blue_brain_project_sdk as bbp
n = bbp.Neuron()
```

3.4 Whitespace in Expressions and Statements

3.4.1 *Always surround binary operators with a single space on either side*

assignment (=):

```
x = y
```

augmented assignment (`+= -= *= /= %= **= <<= >>= &= ^= |=`):

```
x += 1
x >>= 2
x ^= x
```

comparisons (`==, <, >, !=, <>, <=, >=, in, not in, is, is not`):

```
x == y
x < y < z
x in numbers
x is not y
```

Booleans (`and, or, not`):

```
x and y
x  or  y
not x
```

The exception here is the power operator (`**`), which is usually not surrounded by spaces:

```
x = y**2
```

3.4.2 But do not use a space when defining keyword arguments

```
def fun(x, y, size=23, initial_pos=(0, 0)):
    pass
```

3.4.3 Use spaces around arithmetic operators

The following formatting is recommended by PEP8 [3], but maybe violated for sake of readability:

```
# CORRECT
i = i + 1
submitted += 1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)

# WRONG (Maybe)
i=i+1
submitted+=1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Permissible violation:

```
# Indicate operator precedence
hypot2 = x*x + y*y
# Indicate grouping
c = (a+b) * (a-b)
```

3.4.4 *But avoid extraneous whitespace*

Immediately inside parentheses, brackets or braces:

```
spam(ham[1], {eggs: 2})        <-- CORRECT
spam( ham[ 1 ], { eggs: 2 } )  <-- WRONG
```

Immediately before a comma, semicolon, or colon:

```

if x == 4: print x, y; x, y = y, x       <-- CORRECT

if x == 4 : print x , y ; x , y = y , x <-- WRONG
```

Immediately before the open parenthesis that starts the argument list of a function call:

```
spam(1)   <-- CORRECT
spam (1)  <-- WRONG
```

Immediately before the open parenthesis that starts an indexing or slicing:

```
dict['key'] = list[index]   <-- CORRECT
dict ['key'] = list [index] <-- WRONG
```

3.4.5 *Always Follow a comma with a space*

```
x = [1, 2, 3]
y = {"a":1, "b":2}
```

3.5 Naming Conventions

3.5.1 *Packages, Modules and Scripts*

Should always use **lower_case_with_underscores**:

```
package
+--   init  .py
+--module
   +--   init  .py
   +-- code_file.py
```

Some filesystems are case insensitive, so using upper case to distinguish files is highly discouraged. Also, don't use a hyphen (-) in filenames. If you ever need to import that file, the hyphen will be interpreted as the subtraction operator.

3.5.2 *Classes*

Use **CamelCase**

```
class IonChannel(object):
    pass

class PostStimulusTimeHistorgram(object):
    pass
```

PEP8 [3] does not mention anything about capitalisations of acronyms included in class names, but for the BBP the recommendation is to keep the acronym upper case:

```
# CORRECT
class PSTHPlot(object):
    pass

# WRONG
class PsthPlot(object):
    pass
```

3.5.3 Functions and methods

Use **lower_case_with_underscores**

```
def inter_bouton_interval(self):
    pass

def morphology_label(self):
    pass
```

3.5.4 Constants/Literals

Use **ALL_CAPITALS** for anything that would be a `#define` or `const` in C/C++, for example:

```
PI = 3.1459
TIMEOUT = 2.3
```

3.5.5 Variables

Use meaningful but short names for your variables. I.e. avoid single character variables but also overly long and hard to type names:

```
# GOOD
indexes = [1, 2, 3]
# BAD
i = [1, 2, 3]
idx = [1, 2, 3]
indexes_into_array_of_floats_with_widget = [1, 2, 3]
```

The obvious exception here, is when you are writing very mathematical code which is made to look more like written formulas. In this case you may want use inline comments to indicate what your variables mean, or use LaTeX in your docstrings.

3.5.6 Note on the BBP-SDK Python bindings

The bindings use the naming convention **Caml_Case_With_Underscores.** Although PEP8 [3] describes this as **ugly**, you should adhere to this convention, since consistency is better than a mixed style (see above).

```
svd = bbp.Segment_Voxel_Density()
```

3.6 Exceptions

- Use the standard exception hierarchy [4] whenever possible, instead of defining you own custom exceptions.

Here are the most important and useful ones:

```
Exception
+-- StandardError
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
    +-- AssertionError
    +-- IndexError
    +-- TypeError
    +-- ValueError
```

The three most used in user code are `IndexError`, `TypeError` and `ValueError`:

```
class CustomContainer(object):
    """ Custom list which is preallocated and holds only integers. """

    def   init  (self, size):
        if size < 0:
            raise ValueError(
                "Size of CustomContainer may not be less than zero.")
        else:
            # preallocate a list for storage
            self._container = [0 for i_ in range(10)]

    def _check_index(self, index):
        if index >= len(self):
            raise IndexError("CustomContainer index out of range.")

    def get_item (self, index):
        self._check_index(index)
        return self._container[index]

    def set_item (self, index, item):
        self._check_index(index)
        if not isinstance(item, int):
            raise TypeError("CustomContainer can only hold integers."
        self._container[index] = item
```

4 Static Checking

You can (and should) use automated tools to check if your code conforms to the coding conventions (a.k.a. Static Checking):

- pylint [5]
- pyflakes [6]
- pep8-tool [7]
- PyChecker [8]

4.1 Comparison

Of all of these, pylint [5] is by far the most widely used tool. The number of issues that it can detect is quite comprehensive but it takes quite long to run. pyflakes [6] and pep8-tool [7] are nice alternatives that run quicker. The first detects logical errors such as unused imports and redefined names. The second checks code style such as whitespace, correct use of blank lines. However, at the time of writing neither was able to detect adherence to the naming conventions or missing docstrings, which pylint [5] did detect.

PyChecker [8] is the oldest of the tools. It one major disadvantage compared to the other tools is, that it imports code, which may trigger unwanted side effects such as SQL Connections, installation of configurations files etc. Because of this it is not recommended. The other three analyse the abstract syntax tree instead which prevents any side effect.

The recommendation is to use pylint. If this is too pedantic for you or runs too slowly, use a combination of pyflakes [6] and pep8-tool [7]. Note also that since pylint is so pedantic it is usually not feasible to target a score of 10/10. Instead you should use pylint to guide you towards the most severe violations and use good measure to ignore the lee important ones. A pylint score of above 7 is usually sufficient.

There are several ways to integrate automatic checking into your editor of choice, Google is your friend here. For this use-case, pylint [5] may not be that suitable, since it has a long runtime. Therefore pep8-tool [7] and pyflakes [6] are more suitable.

4.2 Example

Consider the following piece of badly written code:

```
import sys, os


lowercase_constant=1
print "import side effect"


class    no_caml_case(object):

    def   init  (self,param1):

        unused_variable=(1,2,3)
        self.param1 = param1
    def getParam1(self):
        return self.param1
```

Running `pylint`:

```
$ pylint badcode_example.py
No config file found, using default configuration
************* Module badcode_example
C:  1: Missing docstring
C:  3: Operator not preceded by a space
lowercase_constant=1

                ^
C:  3: Invalid name "lowercase_constant" (should match (([A-Z_][A-Z0-9_]*)|(  .*  ))$)
C:  5:no_caml_case: Invalid name "no_caml_case" (should match [A-Z_][a-zA-Z0-9]+$)
C:  5:no_caml_case: Missing docstring
C:  7:no_caml_case.  init  : Comma not followed by a space
    def   init  (self,param1):

                    ^^
C:  8:no_caml_case.  init  : Operator not preceded by a space
```

```
        unused_variable=(1,2,3)
                        ^
W:  8:no_caml_case.  init  : Unused variable 'unused_variable'
C: 10:no_caml_case.getParam1: Invalid name "getParam1" (should match [a-z_][a-z0-9_]{2,30}$)
C: 10:no_caml_case.getParam1: Missing docstring
R:  5:no_caml_case:  Too  few  public  methods  (1/2)
W:  1: Unused import sys
W:  1: Unused import os

...

Global evaluation
-----------------
Your code has been rated at -3.00/10 (previous run: -3.00/10)
```

Running pep8:

```
$ pep8 badcode_example.py
badcode_example.py:1:11: E401 multiple imports on one line
badcode_example.py:3:19: E225 missing whitespace around operator
badcode_example.py:5:1: E302 expected 2 blank lines, found 1
badcode_example.py:7:22: E231 missing whitespace after ','
badcode_example.py:10:5: E301 expected 1 blank line, found 0
```

Running `pyflakes`:

```
$ pyflakes badcode_example.py
badcode_example.py:1: 'sys' imported but unused
badcode_example.py:1: 'os' imported but unused
badcode_example.py:8: local variable 'unused_variable' is assigned to but
never used
```

As you can see, at least in the case of this limited example, pylint [5] outperforms the other tools. It even has a single number summary of how good you code is.

Just for reference, here is the output of PyChecker [8], which shows how the *print* statement is triggered:

```
$ pychecker badcode_example.py
Processing module badcode_example (badcode_example.py)...
import side effect

Warnings...

badcode_example.py:1: Imported module (os) not used
badcode_example.py:1: Imported module (sys) not used
```

5 Testing

Unit testing [9] is vital for creating robust Python programs, since we don't have a compiler to check types for us. As with the automatic style checkers we have a multitude of options again, the most important of which are:

- unittest [10]
- nosetest [11]
- py.test [13]

For further reading including a comprehensive list of available tools please consult The Python Testing Tools Taxonomy [14].

The unittest [10] module is part of the standard Python library. The nosetest [11] module must be installed additionally. In Ubuntu the package is called `python-nose`. **When developing software for the Blue Brain Project we recommend that you don't use the unittest module, but instead write your tests exclusively using the nosetest apis and run them using the nosetests command line tool.** Although the burden is an additional dependency, this is pure Python and available for a wide range of distributions and the advantages and work-flow simplifications make it really appealing over unittest [10]. The following sections will describe why. py.test [13] is quite similar to nosetest [11] but does not have as much momentum within the python community.

Important note: many existing projects already use the unittest module. In this case, it is not worth refactoring everything since first, the unittest module is part of the standard library so no additional dependencies are incurred and second, nosetest can handle tests written with unittest.

Lastly, if you are using Numpy, be sure to read the last section which introduces additional mechanisms to test Numpy's arrays.

5.1 Unitest

The unittest [10] module has been the standard for a long time, but suffers from old age. It was initially inspired by the Java unit testing framework JUnit and as a result suffers due to various non-pythonic aspects. Its two major shortcomings are, that it does not adhere to the PEP8 [3] recommendation for function names and requires a lot of boilerplate when writing test:

```
import unittest as ut

class TestSum(ut.TestCase):

    def testSum(self):
        self.assertEqual(sum([1, 2, 3]), 6)

if   name___== '  main  ':
        ut.main()
```

The first thing to note is that the function `assertEqual` does not adhere to the Python recommendation **lower_case_with_underscors**, but instead is **mixedUpperAndLowerCase** as in Java. This means that if you do initially adhere to the python PEP8 [3] recommendation and also use the unittest [10] framework, you tests will unavoidably end up being mixed style. Additionally the `test_sum` function requires two unnecessary and non-pythonic boilerplate snippets. First, a sub-class of `TestCase` must be defined to contain the function. And secondly, the statement `ut.main()` must be included for the unittest [10] framework to discover and run tests in this file.

Running the code:

```
$ python unittest_example.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

As indicated above the statement `ut.main()` causes all tests to be run. The default unittest [10] runner will display a period (`.`) for each successful test but can be customised to provide more verbose output. Obviously, if any tests fail, a traceback is displayed.

5.2 Nosetest

With `nose` there is no need to define a class or a `ut.main()` statement because the command line tool `nosetest` will perform automagic discovery of your tests. As an additional benefit you can use the pythonic `assert_equal` instead of the Java-like `assertEqual`:

```
import nose.tools as nt

def test_sum():
    nt.assert_equal(sum([1, 2, 3]), 6)
```

To run the test you would use the `nosetest` command line tool:

```
$ nosetests nosetest_example.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The `nosetest` command features automatic discovery of all of your tests using clever heuristics. Also, you can filter tests and thus only run the ones your are currently interested in, for example, when fixing a bug or developing a feature. Have a look at the homepage for more information. Furthermore, it is extensible and there is a plug-in for the coverage module [12] to run the tests while simultaneously determining test coverage.

One important switch for `nosetests` is `-v` which will increase the verbosity of the tool such that the test which are also printed to the screen:

```
$ nosetests -v nosetest_example.py
nosetest_example.test_sum ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

5.3 Numpy Testing

Importantly Numpy provides some additional array testing methods [15] which make testing code that uses arrays much easier. Trying to test arrays using the standard way to test for equality of elements:

```
>>> import numpy
>>> import nose.tools as nt
>>>  nt.assert_equal(numpy.zeros(10), numpy.zeros(10))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

/home/haenel/<ipython console> in <module>()

/usr/lib/python2.7/unittest/case.pyc in assertEqual(self, first, second, msg)
    501         """
    502         assertion_func = self._getAssertEqualityFunc(first, second)
--> 503         assertion_func(first, second, msg=msg)
```

```
    504
    505     def assertNotEqual(self, first, second, msg=None):

/usr/lib/python2.7/unittest/case.pyc in _baseAssertEqual(self, first, second, msg)
    491     def _baseAssertEqual(self, first, second, msg=None):
    492         """The default assertEqual implementation, not type specific."""
--> 493         if not first == second:
    494             standardMsg = '%s != %s' % (safe_repr(first), safe_repr(second))
    495             msg = self._formatMessage(msg, standardMsg)

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

If instead we use `numpy.testing`:

```
>>> import numpy.testing
>>> numpy.testing.assert_array_equal(numpy.zeros(10), numpy.zeros(10))
```

Two additional important functions to mention are `assert_array_almost_equal` `assert_almost_equal`. These can assert that two arrays are equal up to a certain precision. This is important when testing numerical code. For example, when using two different numerical optimisers the solution will differ ever so slightly. For example the first algorithm may find the solution at `3.145875634` whereas the second may find it at `3.145875492` This is to be expected due to numerical precision and convergence parameters of the algorithms. In order to convince oneself, that they do actually find the same optimum ones needs to compare the solution only up to a significant digit:

```
>>> numpy.testing.assert_almost_equal(3.145875492, 3.145875634, decimal=6)
>>> numpy.testing.assert_almost_equal(3.145875492, 3.145875634, decimal=7)
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)

/home/haenel/<ipython console> in <module>()

/usr/lib/pymodules/python2.7/numpy/testing/utils.pyc in assert_almost_equal(actual, desired, decimal, err_msg, verbose)
    461         pass
    462     if round(abs(desired - actual),decimal) != 0 :
--> 463         raise AssertionError(msg)
    464
    465

AssertionError:
Arrays   are   not   almost   equal
ACTUAL: 3.145875492
DESIRED: 3.145875634
```

6 Documentation

This section deals with best practices for documentation, that is syntax for docstrings and fileheaders.

6.1 Docstrings

- Use the docstring mechanism to embed documentation into your source code. This ensures that the documentation is available as help from an interactive session and that tools can automatically extract the documentation to build a website.
- A one line docstring is the absolute minimum requirement:

```
def load_neurons(path):
    """ Load the neurons from a circuit at path. """
    pass
```

6.2 Numpy Docstrings

Instead of the standard documentation guidelines [16] please use the Numpy docstring guidelines [17]. This has the advantage of also requiring you to specify the type of the arguments, which is very useful for numerical/scientific code.

Since the standard is quite lengthy we list only the most important aspects here:

- Single line summary
- Detailed description paragraph
- Parameters and Returns section (if applicable) including the types
- Note that for classes, the constructor is documented in the class docstring

```
class SpamHamEggs(object):
    """ An object to hold a numerical breakfast.

    This is a useful object which will hold and mix all of our breakfast.
    The object is able to, amongst other things, to mix and serve a random
    breakfast.

    Paramters
    ---------
    spam : float
        the yummy bit
    ham : list
        the delicate bit
    eggs : ndarray
        the mushy bit
    """

    def   init  (self, spam, ham, eggs):
        pass

    def random_mix(self, num_desired):
        """ Mix the components

        Parameters
        ----------
        num_desired : int
            the number of components

        Returns
        -------
        mix : list
            a list containing the number of desired components in a random
            order

        """
        pass
```

For a full blown example, see the numpy docstring example [18]

6.3 File Header

Each Python file should have a header of the form:

```
""" Brief summary of the functionality of this file/module.
Optionally, an extended summary of the functionality of this file.

@author  Valentin Haenel

@remarks Copyright (c) BBP/EPFL 2005-2012; All rights reserved.

         Do not distribute without further notice.

"""
```

The initial author and the copyright are important. However it is not necessary to modify this header when changing a file, or including `@date` or `@filename` tags. Instead you should use the version control system to figure out who has worked on the file and when it was last modified.

6.4 Automatic API Generation

Unfortunately there are a number of different tools for automatically generating API documentation from docstrings in the source code and the Python community has not yet converged on one standard. Hence for the time being, please adhere to the numpy docstring guidelines. This will ensure, that the future tool of choice will be able to deal with your software. Note also, that doxygen supports extracting docstrings out-of-the-box with the two caveats that doxygen's special commands such as *@param* or *@var* do not work and that doxygen is not able to render reStructuredText.

7 Writing the `setup.py` file

The `setup.py` is the quasi equivalent of a `makefile` or `build.xml`. The big difference is that, since Python is an interpreted language, no compilation is required and hence the build file is quite minimal. Its main use is to contain metadata, for example the author and copyright information, as well as information about the version and any modules the software provides. This information can be used to install the software in the correct place. The main functionality needed in the `setup.py` file is provided by the `distutils` module. For more information see the guide to writing the setup file [19].

Also the `setup.py` has some command line help:

```
$ python setup.py --help
Common commands: (see '--help-commands' for more)


setup.py build      will build the package underneath 'build/'
    setup.py install    will install the package
...


$ python setup.py --help-commands
Standard commands:

build             build everything needed to install

build_py          "build" pure Python modules (copy to build directory)
build_ext         build C/C++ extensions (compile/link to build directory)
...
```

The `setup.py` may also contain information about extensions written in compiled languages. This is particularly important for scientific software since a layered approach that combines high performance code in a compiled language with an easy to script python interface is extremely common. In this case the `setup.py` can be used to specify which sources files belong to extension. Although the distutls module can compile such extensions, it does suffer from some major shortcomings, the most grave being an inability to detect what needs to be recompiled. This means, that whenever you wish to re-compile your extension, the entire code needs to be recompiled, as opposed to using `make` which supports re-compiling only those files which have changed. If compilation time is an issue it may make sense to craft a buildsystem using a combination of CMake and a `setup.py`.

At the blue brain project pure python programs should contain a minimal version for this file. For example:

```
#!/usr/bin/env python

from distutils.core import setup

setup(name='bbp-example-toolkit',
    version='1.5.3',
    description='Example Toolkit for the Blue Brain Project',
    author='Valentin Haenel',
    author_email='valentin.haenel@epfl.ch',
    url='http://bbpteam.epfl.ch/documentation/bbp_example',
    packages=['bbp_example'],
    )
```

The Debian packaging system has the ability to hook into a `setup.py` and use the native tools provided to build a package. This can save a lot of time and hassle when developing a Debian package for a Python tool.

8 Versioning

A software should have a version number which is a triplet of `MAJOR.MINOR.PATCH`, for example `0.2.4`. Compared to C++ where we need to be careful to maintain binary compatibility, we can be somewhat less strict in Python. Use the following guidelines when deciding how to increment the version: The `PATCH` version number should be increased for changes that do not alter the public API of your system. I.e. for those where no refactoring is required for any software that uses your software; The `MINOR` number should be increased in case of public API changes. That is to say, software that uses your software will have to be refactored minimally to work with the new version; The `MAJOR` number should be increased when you overhaul your software and make sweeping changes to the public API and internals. In this case any software using your software may have to undergo substantial changes to work with the latest version.

You may also want to look at the versioning guideline for BBP software [20] and the Wikipedia article about versioning [21]

The version should be stored in a single place in your software, for example in a file `version .py` in your package, i.e:

```
mypackage
+--   init  .py
+--   version  .py
```

Where the contents of `version  .py` is for example:

```
version = "0.1.2"
```

And `init  .py` contains:

```
from   version___import version
```

You can then access the version number using:

```
>>> import mypackage
>>> mypackage.version
'0.1.2'
```

You may also want to read the *stackoverflow article* about this, which suggests somewhat more sophisticated techniques for storing the version number.

9 Obtaining Python Packages

This section explains the best strategies to follow when installing Python software on your system.

9.1 Python on Linux

When using Linux, the preferred method to obtain and install Python packages is using the package manager. In Debian/Ubuntu this means using one of `apt-get`, `aptitude`, `synaptic` or any other front-end to `dpkg`. This is by far the easiest and most user-friendly way. Dependencies are installed automatically, software (and its dependencies) can be removed easily and software is easily upgraded. **Before looking at any installation documentation, check if the desired software is available via the package manager**

For example to install `nosetest`:

```
$ sudo aptitude install python-nose
```

In certain cases it may not be feasible to use the package manager. For example, if the software you are looking for has not yet been packaged for your target distribution, when you are using an exotic system which may only have old and outdated packages or if you do not have sufficient privileges to install the software system-wide.

In such cases you may be able to install it from the Python Package Index (PyPi [22]) using either pip [23] (preferred) or easy_install [24] (if `pip` is not available. The one major advantage that pip [23] has over easy_install [24] is that it can uninstall packages too. However bear in mind, that the package manager of your distribution will not be aware of packages installed this way. For example to install the nosetest package `nose` available from PyPi [22]:

```
$ pip install --user nose
$ easy_install --user nose
```

Using the `--user` flag will install the software in the users local site-packages directory, for example `/home/haenel/.local/lib/python2.7/site-packages`. This has the advantage, that the directories are automatically added to your `$PYTHONPATH` and no further manipulation of this variable is needed.

Note

Versions of `pip` prior to `0.8.1` did not directly support the `--user` option. Instead you have to use `--install-option="--user"`, for example:

Since pip [23] is pure python it is easy to install, detailed instructions can be found on their website.

The last resort for obtaining software is downloading the source code and doing a manual install via a (hopefully) provided `setup.py` file or use of the `$PYTHONPATH` environment variable. This variable contains paths which are searched by the Python interpreter when attempting to import modules. For more information about the `setup.py` file, see the section: Writing the setup.py file.

For example, if you have downloaded the package `foobar` which is pure Python code (i.e. no C or FORTRAN extensions), to the directory `$HOME/src/foobar`. It may be enough to set:

```
PYTHONPATH=$HOME/src/foobar
```

... to make the software available.

If the software has a proper `setup.py` file, you can use this to install or build extensions.

Although many websites and forums suggest simply running:

```
$ python setup.py install     # <-- BAD PRACTICE
```

... this is frowned upon by many experienced developers. The problem is, that this simply copies the respective files somewhere into your operating system, which may make them hard to find or delete. It is a similar problem as using pip [23] or easy_install [24]. There exists several ways to install the software in a user-defined location. The easiest is:

```
$ python setup.py install --user
```

The `--user` flag works here too as described above, and hence there is no further need to modify the `$PYTHONPATH` variable.

9.2 Python on Windows and Mac

Enthought [25] provides the Enthought Python Distribution (EPD [26]) which is available for download for strictly academic uses [27]. It provides a large number of software including the most important packages for scientific computing, Numpy, Scipy and Matplotlib. It is available for Windows, Mac OSX, Linux and Solaris. An alternative for Windows is the Python(x,y) [28] distribution which also included the scientific computing packages, but seems to be available for windows only.

10 Wrapping Native code

One of Pythons greatest strengths is that it allows you to wrap native code written in C/C++ and FORTRAN. This has several interesting use-cases for scientific computing; first of all it is feasible to take existing legacy code and make it available through an accessible Python interface; secondly, code that has a high run-time can be implemented in C/C++ for speed and made accessible in Python for convenience.

Unfortunately a number of very different options exist to achieve this goal and describing all of them is well beyond the scope of this document. At the Blue Brain Project we make heavy use of the boost.python [29] library, especially for the BBP-SDK. Also there is some limited use of cython [30]. If a software project is already using one or the other technology please continue to use this, if you need to modify or extend the wrapping.

Since developing wrappings depends largely on the problem at hand, the required scope of the wrapper and the development status of the wrapping tools, it is not really possible to make any general recommendation. The advice here is that, if you need to develop wrappers consult with the engineering team and the general project manager to decide the best approach.

11 Performance Considerations

Often people complain about Python being too slow. Hence, the section is a small overview of performance related issues in Python. Often these comments are unjust and mal-informed. Hence this sections aims to give a little insight into performance considerations and optimization for Python, to make sure you don't fall victim to the many pitfalls.

While reading this section, bear in mind, that Python performance optimization techniques and approaches are still an active area of research and also that optimization is considered somewhat of an black art amongst computer scientists, often cited as one of the hardest tasks in computing. Hence it pays to be humble when optimizing -- speak to your colleagues and discuss ideas.

Two good links for the interested reader are the page about Pythons profilers [31] and the Scipy page on PerformancePython [32].

11.1 Define "Too Slow"

To begin with, one needs to define what "too slow" actually means in your context, since it is often enough a measure that relies on your own perception. Does too slow mean your simulation runs for 2 months, but you want to your results in one month? Or does it perhaps mean, you don't wish to "get a coffee" every time your program computes new results?

11.2 Identifying the Bottlenecks

Usually, Python programs are fast enough for the task at hand. If they are provably not there are several approaches to reducing the run-time. All of them hinge on the use of a so-called profiler. This is software tool which allows you to inspect the run-time of your code according to its constituent parts. For example, a profiler will tell you how often a function was invoked during a run of your software, and how long this invocation took on average. This then allows you to identify clearly the bottlenecks of your application. Now you can target exactly those bottlenecks and thereby improve the overall performance of your application.

11.2.1 Improve your Algorithm

Pose the fundamental question: Can your algorithm or its implementation do better? Is there a for-loop somewhere that can be avoided. Can you use a hash-table to do look-up instead of a sorted-list? Can you use a tree data-structure instead of linked-list? Many famous basic algorithms for tasks such as sorting and searching or graph traversal were improved over the years, maybe you can find a shortcut in yours too?

11.2.2 Using Numpy

If you need to do anything with vectors or matrices use numpy [33]. This is a very widely used, stable and well documented library for numerical work. It is written in C using Python's native C interface which usually results in a speed increase of an order of magnitude when compared to pure python. It provides an N-dimensional array object which is particularly well suited for matrix and vector math.

11.2.3 Implement your Bottleneck in C

If you discover, that you bottleneck can not be optimized by improving your algorithm or vectorizing your code using numpy, the last resort is to implement a C extension of your bottleneck. This should be the last resort, because it is a fragile process that requires compiling code. There are several approaches here, but many people have reported good success with cython [30] in the past and we recommend this for partial implementation of your program in C.

11.2.4 *Parallelize your Code*

An alternative to implementing you bottleneck in C is to parallelize your code. Of course, this is only relevant if you have available resources such as a graphics card, a compute cluster or a supercomputer. Unless your problem can be easily split into several parts parallelizing it is a highly non-trivial task.

11.3 Avoiding Premature Optimization

Avoid the pitfall of assuming that your application will be too slow and assuming that you will have to write it in C for it to work at all. In this case you may want to consider implementing a prototype in Python (Python is written quickly after all) to investigate the feasibility of your idea *before* considering an implementation in native code. Perhaps Python is fast enough after all.

11.4 Consider time for Development

The cost of developing native code when compared to python is significant. Python has several advantages which seriously reduce development time, such as: automatic memory management, a large standard library and a big offering of third-party libraries for many purposes. Often development time is not considered when estimating the speed of an application. This is especially relevant for the scientist, who needs to produce results quickly. In this case a not-so-fast Python application which takes 3 days to develop is often better than a super-fast C application which takes 3 weeks to develop.

12 Target Version

This section describes what version of Python to code for.

12.1 Python 2.x

There are currently several Python versions in use across distributions:

Distribution/Server	Python Version
Debain Stable	Python 2.6.6
Ubuntu Natty	Python 2.7.1+
LinServ1	Python 2.4.3
LinServ2	Python 2.6.5
bbpdbsrv2	Python 2.4.3
bbpsg1	Python 2.5
BlueGene	Python 2.4 / 2.6 `(Special)`

Although the lowest common denominator is 2.4, I would still recommend to target 2.6, since the older machines are being phased out anyway. It's available on linserv2 and there are packages available in Ubuntu, although its not the default version there.

Python on the BlueGene is somewhat special. If you are developing for or porting too this platform please speak to Eilif Muller and check Confluence for additional documentation on this topic.

12.2 Python 3.x

The up-and-coming version is Python 3000, also known as Py3k or Python 3.0. This has some major improvements which remove long standing cruft and uglyness, but are largely backward incompatible with the 2.x line. You may want to look at What's New In Python 3.0 [34] There is a useful Python2orPython3 FAQ [36] which describes the differences, has some recommendations about what version to use under what circumstances and also contains many links and pointers.

Although there is an automated tool called 2to3 [35] which can convert, it makes sense to follow a few simple rules to future proof your python code [38].

You can import certain changes from future python versions using the future module [37]. The following example demonstrates how to do this using integer division as an example, since this is one very subtle backwards incompatible change which can not be converted or detected by the 2to3 [35] tool.

Historically, the division operator (`/`) in Python has two modes. If both arguments are integers an integer division, also known as *floor division*, is performed. If however, at least one of the arguments is a float, the operator will perform a *true division* and return a reasonable decimal approximation:

```
$ python2
>>> 1/2
0

>>> 1/2.0
0.5

>>> 1.0/2
0.5

>>> 1.0/2.0
0.5
```

This argument-type-dependent behaviour, so called *mixed division*, caused many headaches for new programmers, since the result could be ambiguous or even unpredictable in numerical code. Traditionally, the solution has been to explicitly cast your arguments to `floats` to ensure that the *true division* would be used:

```
>>> float(1)/float(2)
0.5
```

The enhancement proposal PEP238 [39] addressed this issue and it was decided to fix this ambiguous behaviour in Python 3.0. In a nutshell the `/` operator will do *true division*:

```
$ python3
>>> 1/2
0.5

>>> 1/2.0
0.5

>>> 1.0/2
0.5

>>> 1.0/2.0
0.5
```

and the `//` operator will do *floor division*:

```
$python2
>>> 1//2
0

>>> 1.0//2
0.0

>>> 1//2.0
0.0

>>> 1.0//2.0
0.0
```

If you want to write future proof code you can use the aforementioned `future` module to import the *true division* behaviour for the `/` operator into Python 2.x and make sure to use the `//` operator, which has existed in Python 2.x for quite sometime, if you really do want *floor division*:

```
$ python2
>>> from   future___import division

>>> 1/2
0.5

>>> 1//2
0
```

This ensures, that all of your division operations will continue to function as you intended, when you code is ported to Python 3000.

1	http://artificialcode.blogspot.com/2009/08/10-ways-to-let-people-know-your-bad.html
2	http://ipython.org/
3	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) http://www.python.org/dev/peps/pep-0008/
4	http://docs.python.org/library/exceptions.html
5 (1, 2, 3, 4, 5)	http://www.logilab.org/857
6 (1, 2, 3, 4)	https://launchpad.net/pyflakes
7 (1, 2, 3, 4)	https://github.com/jcrocholl/pep8/
8 (1, 2, 3)	http://pychecker.sourceforge.net/
9	http://en.wikipedia.org/wiki/Unit_testing
10 (1, 2, 3, 4, 5, 6, 7)	http://docs.python.org/library/unittest.html
11 (1, 2, 3)	http://readthedocs.org/docs/nose/en/latest/
12	http://pypi.python.org/pypi/coverage
13 (1, 2)	http://pytest.org/latest/
14	http://packages.python.org/testing/
15	http://docs.scipy.org/doc/numpy/reference/routines.testing.html
16	http://www.python.org/dev/peps/pep-0257/
17	https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt
18	https://github.com/numpy/numpy/blob/master/doc/example.py
19	http://docs.python.org/distutils/setupscript.html`
20	https://bbpteam.epfl.ch/confluence/download/attachments/3412334/Versioning.pdf?version=3&modificationDate=1321519
21	http://en.wikipedia.org/wiki/Software_versioning
22 (1, 2)	http://pypi.python.org/pypi
23 (1, 2, 3, 4)	http://www.pip-installer.org/en/latest/
24 (1, 2, 3)	http://packages.python.org/distribute/easy_install.html
25	http://www.enthought.com/
26	http://www.enthought.com/products/epd.php
27	http://www.enthought.com/products/edudownload.php
28	http://code.google.com/p/pythonxy/
29	http://www.boost.org/doc/libs/1_48_0/libs/python/doc/
30 (1, 2)	http://cython.org/
31	http://docs.python.org/library/profile.html
32	http://www.scipy.org/PerformancePython
33	http://numpy.scipy.org/
34	http://docs.python.org/py3k/whatsnew/3.0.html 35(1, 2) http://docs.python.org/library/2to3.html
36	http://wiki.python.org/moin/Python2orPython3
37	http://docs.python.org/library/ future .html
38	http://wiki.python.org/moin/FutureProofPython
39	http://www.python.org/dev/peps/pep-0238/

Annex E – UI Development and Testing Standards

UI Development Guidelines

Author: Martina Schmalholz, Annapaola Santarsiero and Jeff Muller

Date: 2016-04-15

Overview

This document describes the basic process used for the development of UIs in the HBP, based on a standard Agile model. The initial structure is followed by a concrete example from the Collaboratory and SP10 platform development history.

Best practice process follows roughly these steps:

1. concept w/ users
2. use cases
3. optional user validation
4. mockup
5. optional user validation
6. front-end prototype -> dev or staging servers
7. Usability testing by sprint team
8. Refinement -> dev or staging servers
9. Deployment to production servers
10. Feedback from users
11. Optional return to step 6 if needed.

Parts 1-3 are typically done before a sprint and items 4-9 would be attempted in a single 2 week sprint (usually 1-3 UI modifications of this sort happening per sprint). 10-11 would be post-sprint activities.

HBP Collaboratory

The Collaboratory (Collab) is the central hub and main access point to the Platforms. Therefore, there is particular attention in the development of the UI.

User tests are usually executed internally to the development team, or involving users from other projects.

The following example shows the process used for the development of a new functionality in the system that allows to manage groups and members.

Step 1 – Concept

The collab is not only the place where one may access the different features and tools of the platforms, but it also allows for retrieval of more information, and management of the different collabs, apps, and tools.

Groups are collections of identities which are used for various purposes:

- Authorisation – some HBP Platform systems allow operations only by certain groups
- Communication – some HBP Platform systems send communications to a group
- Organisation – some HBP structures need to documented in a database to allow efficient coordination of the project.

The Groups service has been in operation for some time, but there is no way for users to easily edit group mappings. Admin users, in particular, want to decide what resources can be used by new users. The planned way of doing this is by adding them to a Admin define group. The UI will make this a more accessible operation.

Step2 – Description of users needs

Users need a place where they may access information about groups and users and, in case they have admin rights, to manage groups.

To accomplish those needs the following scenarios were defined:

1. As a user, I want to search for a group;
2. As a user, I want to access a group details;
3. As a user, I want to view the list of members and admin(s) of a group;
4. As an admin of a group, I want to add a user or a subgroup as member of a group;
5. As an admin of a group, I want to delete a user from the group members;
6. As an admin of a group, I want to add a user or a subgroup as an admin of a group;
7. As an admin of a group, I want to delete a user from the group admins;
8. As a user, I want to create a new group;
9. As an admin of a group, I want to edit the group name and description; and
10. As an admin of a group, I want to delete a group.

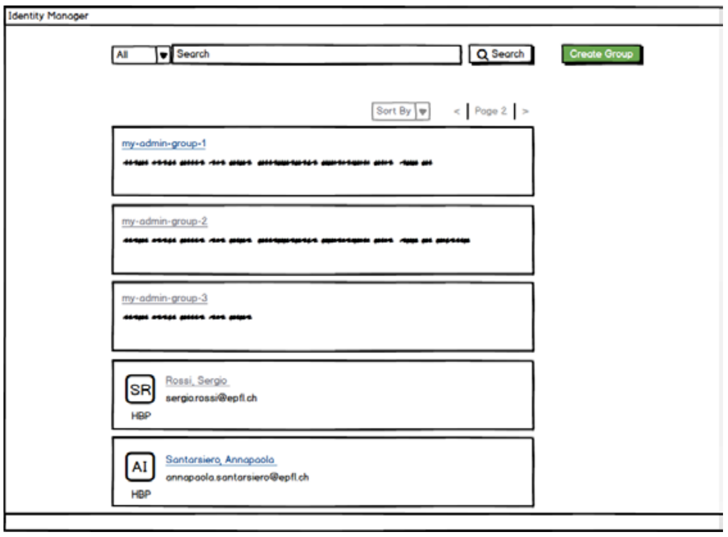
Step3 – User validation

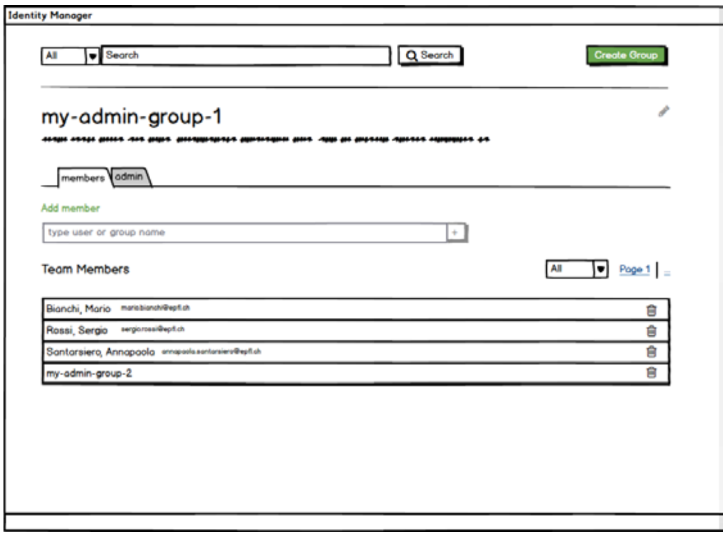
Before starting a sprint, the scenarios are approved by the development team.

Step4 – Mockups

Once the scenarios are validated internally, some mockups are created so they can be shown to users for a further validation.

For brevity, we will focus on the scenario 3 [As a user, I want to view the list of members and admin(s) of a group].





Bianchi, Mario	mario.bianchi@epfl.ch
Rossi, Sergio	sergio.rossi@epfl.ch
Santarsiero, Annapaola	annapaola.santarsiero@epfl.ch
my-admin-group-2	

Step 5 – User validation

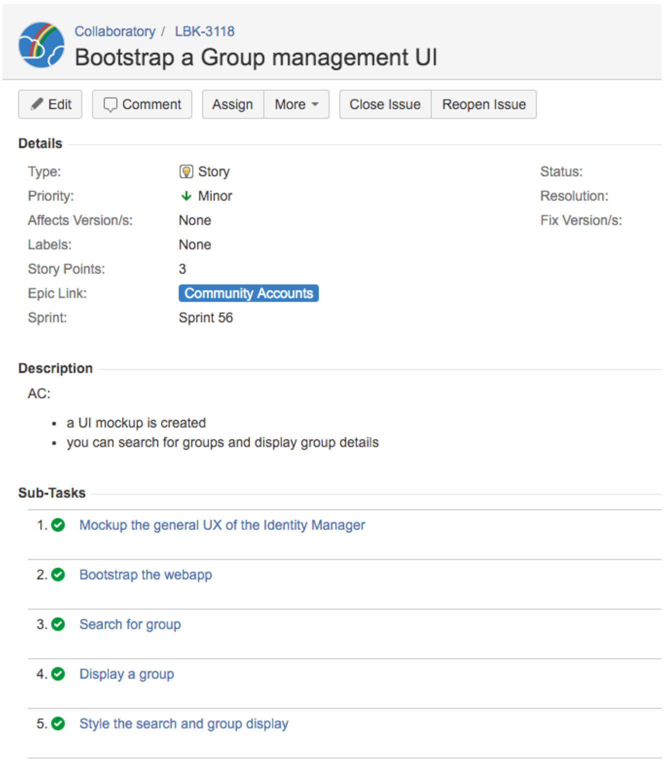
A user test on the group UI is conducted with potential users using interactive mockups with the main purpose of validating the scenarios and the interface.

During the session, which last approximately 10/15 minutes, the participants are asked to follow some scenarios and to comment on them.

Comments are annotated and, if necessary, changes to the mockups are made.

Step 6 (part 1) – Enter stories in the backlog

After validating the mockups, the development work can start. The first step consists in defining the user stories and inserting them in the backlog.



Collaboratory / LBK-3118

Bootstrap a Group management UI

Details

Type:	Story	Status:	
Priority:	Minor	Resolution:	
Affects Version/s:	None	Fix Version/s:	
Labels:	None		
Story Points:	3		
Epic Link:	Community Accounts		
Sprint:	Sprint 56		

Description

AC:

- a UI mockup is created
- you can search for groups and display group details

Sub-Tasks

1. ✓ Mockup the general UX of the Identity Manager
2. ✓ Bootstrap the webapp
3. ✓ Search for group
4. ✓ Display a group
5. ✓ Style the search and group display

Step 6 – Front end prototype

At this point the team can start the development of the UI, which will be deployed on the development site at the end of the sprint.

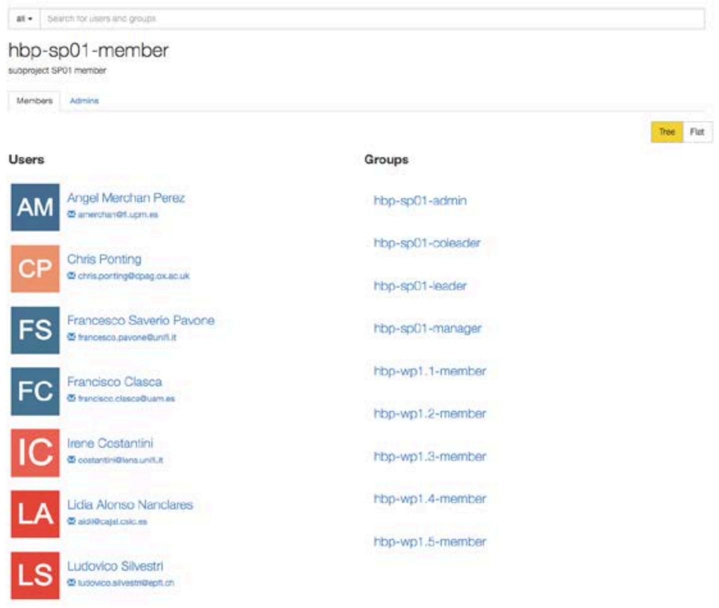
Step7, 8 – Testing and Refinement

During the "end of sprint" meeting, the UI is demoed to the rest of the group and, in some cases, to other users as well.

During the demo, comments are taken in account and used to define a new story or to detail the current one, which will go under further development in the following sprint.

However, if the acceptance criteria are met, the story is marked as "Resolved" and closed.

Below the screenshots of the first version of the group interface shown at the end of the sprint.



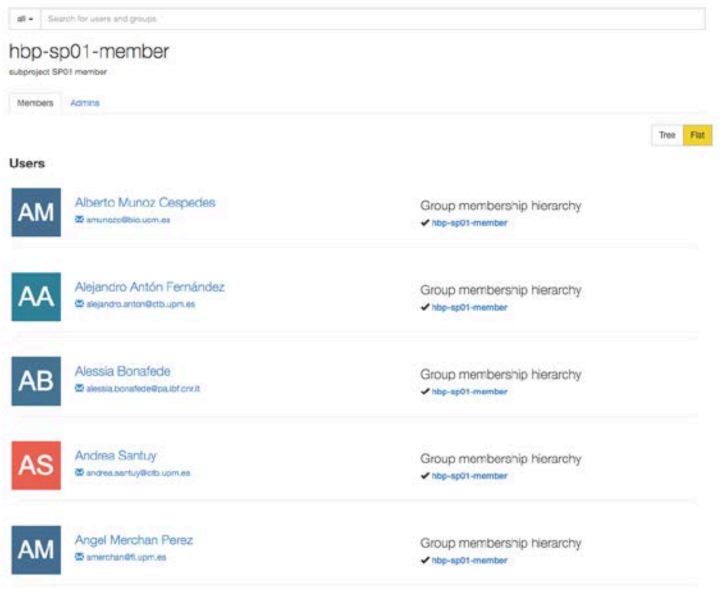
hbp-sp01-member
subproject SP01 member

Users

- AM Angel Merchan Perez — amerchan@fi.upm.es
- CP Chris Ponting — chris.ponting@dpag.ox.ac.uk
- FS Francesco Saverio Pavone — francesco.pavone@unifi.it
- FC Francisco Clasca — francisco.clasca@uam.es
- IC Irene Costantini — costantini@lens.unifi.it
- LA Lidia Alonso Nanclares — aidil@cajal.csic.es
- LS Ludovico Silvestri — ludovico.silvestri@epfl.ch

Groups

- hbp-sp01-admin
- hbp-sp01-coleader
- hbp-sp01-leader
- hbp-sp01-manager
- hbp-wp1.1-member
- hbp-wp1.2-member
- hbp-wp1.3-member
- hbp-wp1.4-member
- hbp-wp1.5-member



hbp-sp01-member
subproject SP01 member

Users

AM Alberto Munoz Cespedes — amunozc@bio.ucm.es	Group membership hierarchy ✓ hbp-sp01-member
AA Alejandro Antón Fernández — alejandro.anton@ctb.upm.es	Group membership hierarchy ✓ hbp-sp01-member
AB Alessia Bonafede — alessia.bonafede@pa.ibf.cnr.it	Group membership hierarchy ✓ hbp-sp01-member
AS Andrea Santuy — andrea.santuy@ctb.upm.es	Group membership hierarchy ✓ hbp-sp01-member
AM Angel Merchan Perez — amerchan@fi.upm.es	Group membership hierarchy ✓ hbp-sp01-member

Step 9 - Deployment on production

If the UI passes the test during the demo, it is then deployed in production so that can be available to all the users.

Subsequent steps

Once released, users can take advantage of the functionality which has been developed. If they have concerns or questions, these are routed through the standard “Feedback” mechanism on the site.

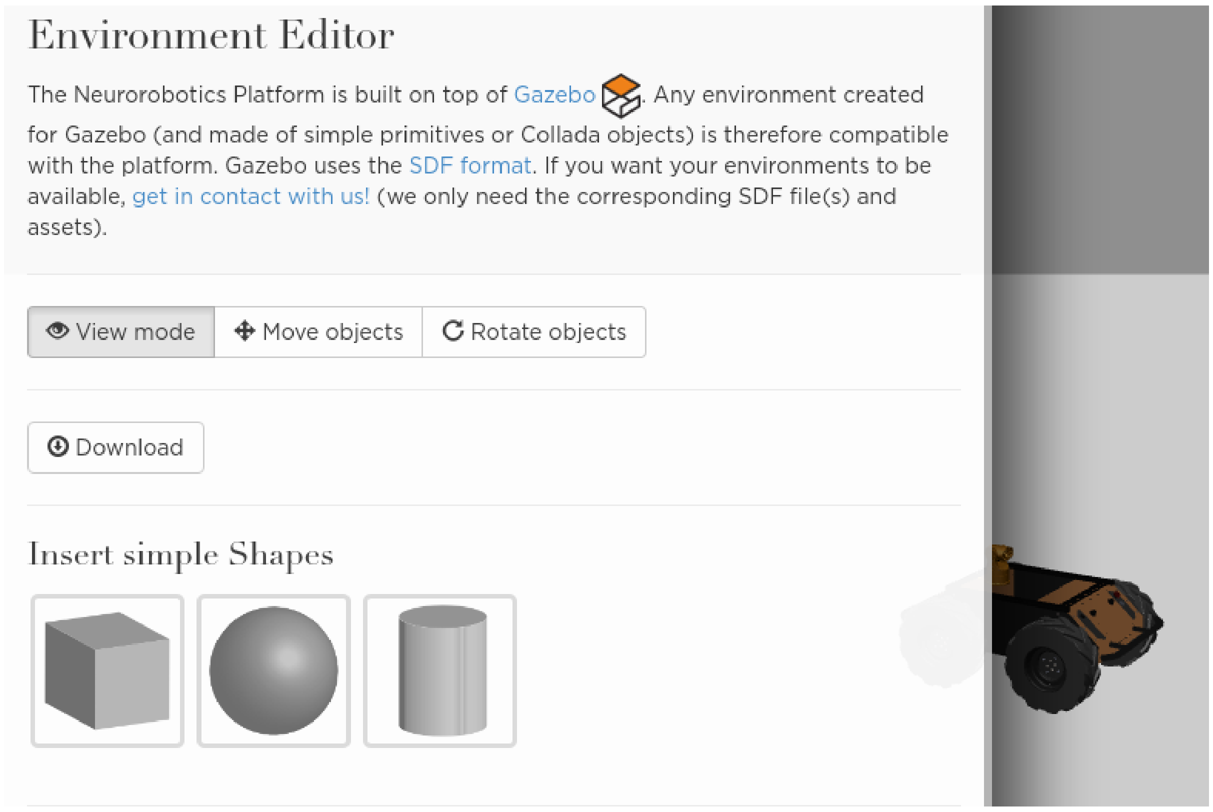
SP10 Example

Human-computer-interface or UI (User Interface) are very important for SP10. In order to provide the best possible user experience, this is how SP10 proceeds for each new UI development according to the process (1-11) outlined above.

Since it is very difficult to have the perfect UI from the beginning, we use the scrum process to have several iterations on a UI element when the users are not satisfied.

The following concrete story shows the process:

Step 1 – Concept (driven by User need)



Environment Editor

The Neurorobotics Platform is built on top of Gazebo. Any environment created for Gazebo (and made of simple primitives or Collada objects) is therefore compatible with the platform. Gazebo uses the SDF format. If you want your environments to be available, get in contact with us! (we only need the corresponding SDF file(s) and assets).

Steps 2 and 3 – Description of User need

The problem:

to switch between view, move and rotate mode for a given object of the scene, the user has to:

- select the object by clicking on it
- open the environment editor
- click on move (for example)
- close the environment editor

- move the object
- reopen the environment editor
- click on view
- reclose the environement editor

This process is absolutely not user friendly. The user does not want to open the environment editor to move an object.

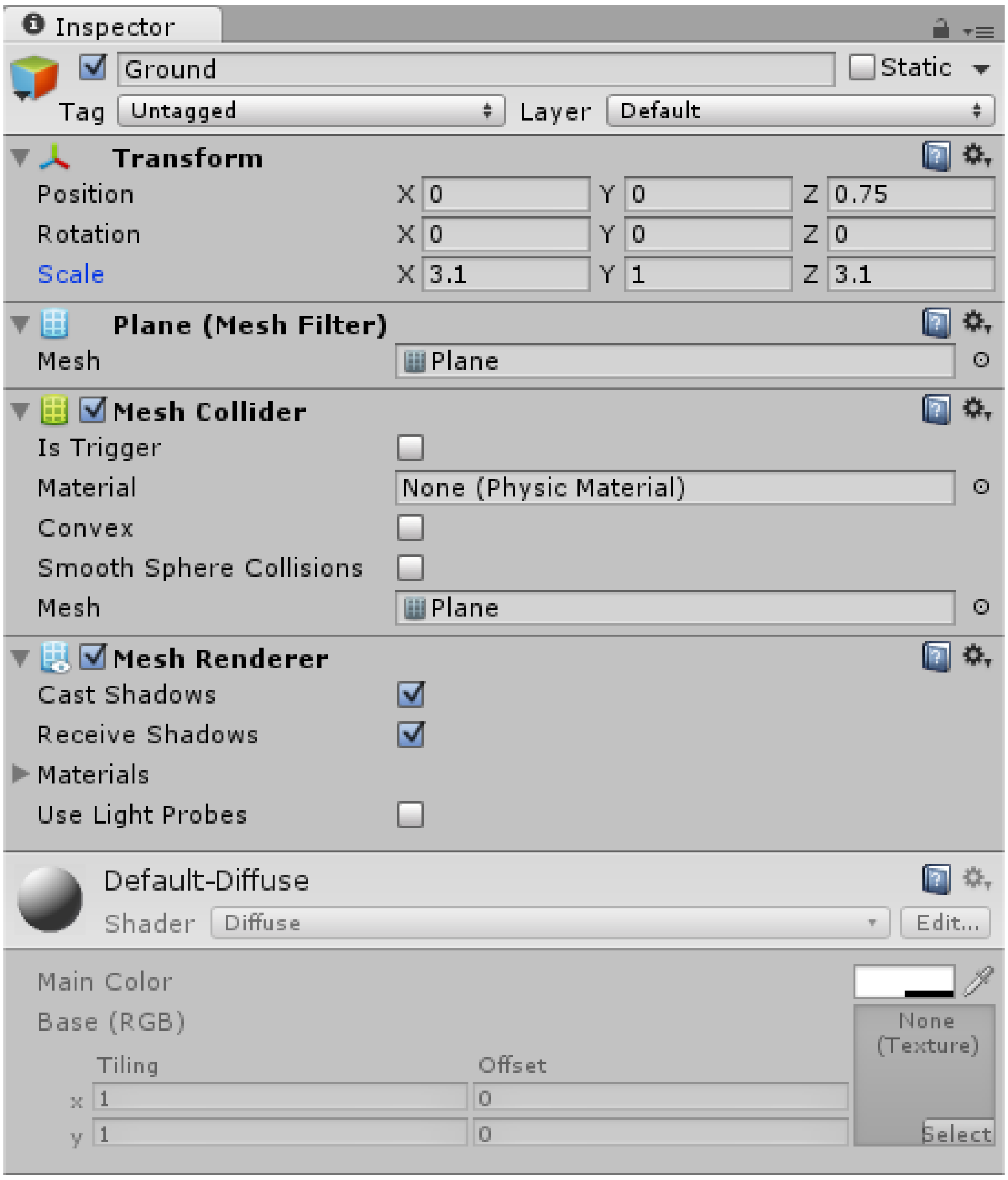
The solution:

The user wants to just right-click on the object and pop up a menu with a sort of floating object inspector as in the mockup.

The move (translate) and rotate button should be packed in this inspector floating panel, as well as the new wished feature to display the object

in wireframe or transparent mode.

Step 4 - mockup



Inspector
☑ Ground ☐ Static
Tag: Untagged Layer: Default

Transform
Position X 0 Y 0 Z 0.75
Rotation X 0 Y 0 Z 0
Scale X 3.1 Y 1 Z 3.1

Plane (Mesh Filter)
Mesh: Plane

☑ Mesh Collider
Is Trigger ☐
Material: None (Physic Material)
Convex ☐
Smooth Sphere Collisions ☐
Mesh: Plane

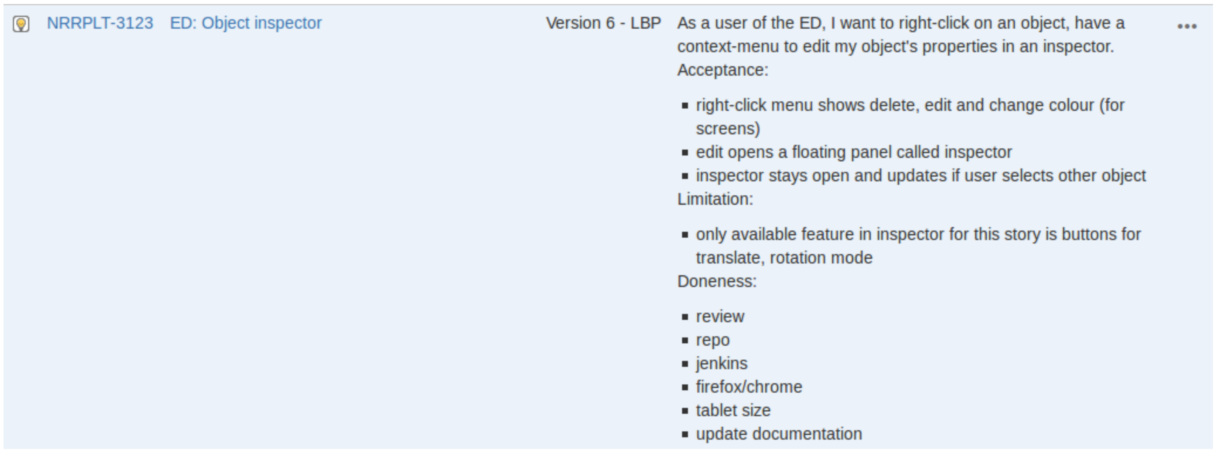
☑ Mesh Renderer
Cast Shadows ☑
Receive Shadows ☑
Materials
Use Light Probes ☐

Default-Diffuse
Shader: Diffuse Edit...

Main Color
Base (RGB) None (Texture) Select
Tiling x 1 y 1
Offset x 0 y 0

In this case SP10 uses an existing piece of software to validate the functionality and UI arrangement. After validating the mockup with representative uses the development work can be broken down in to stories for entry into multiple stories for entry into the backlog.

Step 6 – Part 1 – enter story 1 into the backlog



NRRPLT-3123 ED: Object inspector Version 6 - LBP

As a user of the ED, I want to right-click on an object, have a context-menu to edit my object's properties in an inspector.
Acceptance:

- right-click menu shows delete, edit and change colour (for screens)
- edit opens a floating panel called inspector
- inspector stays open and updates if user selects other object

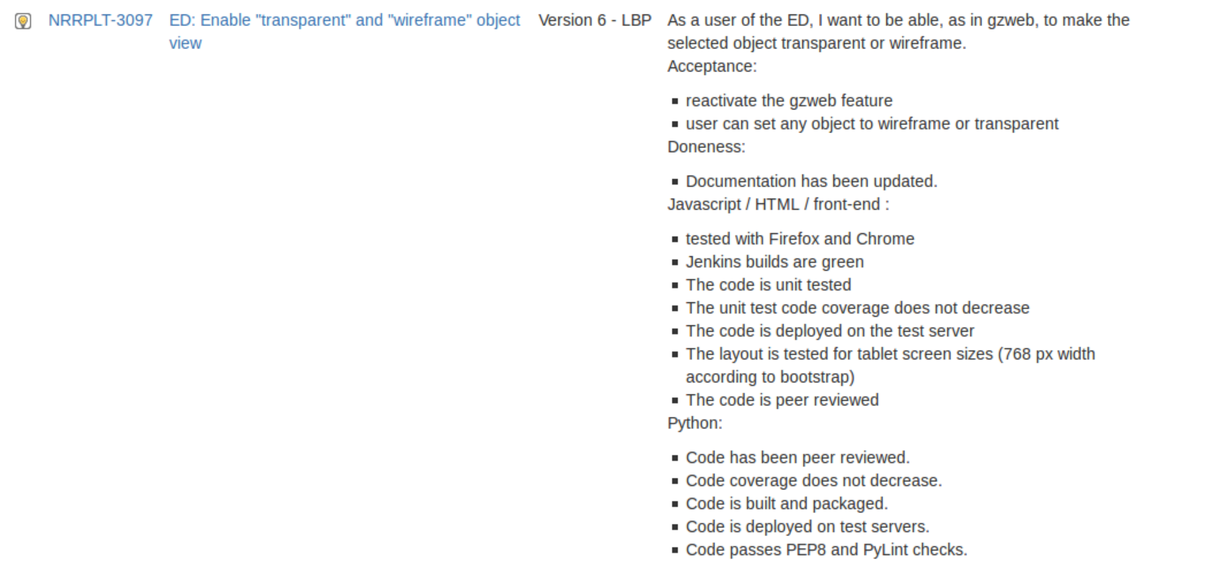
Limitation:

- only available feature in inspector for this story is buttons for translate, rotation mode

Doneness:

- review
- repo
- jenkins
- firefox/chrome
- tablet size
- update documentation

Step 6 – Part 1 – enter story 2 into the backlog



NRRPLT-3097 ED: Enable "transparent" and "wireframe" object view Version 6 - LBP

As a user of the ED, I want to be able, as in gzweb, to make the selected object transparent or wireframe.
Acceptance:

- reactivate the gzweb feature
- user can set any object to wireframe or transparent

Doneness:

- Documentation has been updated.

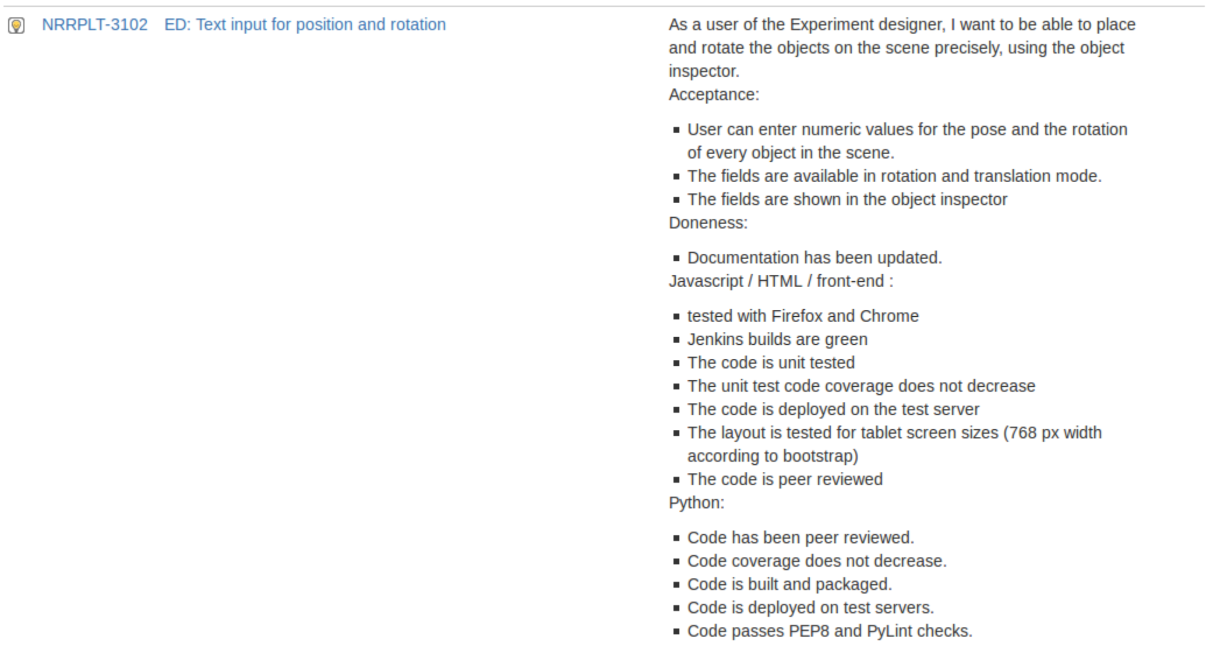
Javascript / HTML / front-end :

- tested with Firefox and Chrome
- Jenkins builds are green
- The code is unit tested
- The unit test code coverage does not decrease
- The code is deployed on the test server
- The layout is tested for tablet screen sizes (768 px width according to bootstrap)
- The code is peer reviewed

Python:

- Code has been peer reviewed.
- Code coverage does not decrease.
- Code is built and packaged.
- Code is deployed on test servers.
- Code passes PEP8 and PyLint checks.

Step 6 – Part 1 – enter story 3 into the backlog



NRRPLT-3102 ED: Text input for position and rotation

As a user of the Experiment designer, I want to be able to place and rotate the objects on the scene precisely, using the object inspector.
Acceptance:

- User can enter numeric values for the pose and the rotation of every object in the scene.
- The fields are available in rotation and translation mode.
- The fields are shown in the object inspector

Doneness:

- Documentation has been updated.

Javascript / HTML / front-end :

- tested with Firefox and Chrome
- Jenkins builds are green
- The code is unit tested
- The unit test code coverage does not decrease
- The code is deployed on the test server
- The layout is tested for tablet screen sizes (768 px width according to bootstrap)
- The code is peer reviewed

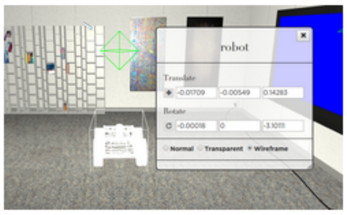
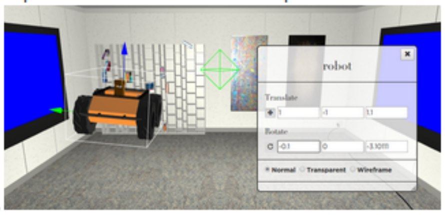
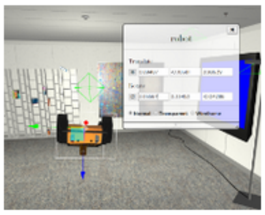
Python:

- Code has been peer reviewed.
- Code coverage does not decrease.
- Code is built and packaged.
- Code is deployed on test servers.
- Code passes PEP8 and PyLint checks.

Step 6,7,8 – In sprint usability testing and review

SP10 tests UIs as they are developed. If the criteria for success are unclear at the point of testing test users are engaged during the sprint to provide immediate feedback.

As UI stories are completed, they are reviewed and recorded in the review report. Typically, this includes a screenshot of the UI in question and some statements about what was implemented. This marks the point at which an iteration of a UI is considered finished. It will then be released to real users for "real-world" feedback.

NRRPLT-3097	ED: Enable "transparent" and "wireframe" object view Sandro implemented different optional renderings of objects in the 3D scene, namely transparent and wireframe renderings. These renderings are triggered from the Object inspector which show when right-click on the 3D object.  robot Translate: -0.01709, -0.00549, 0.14283 Rotate: -0.00018, 0, -3.10111 Normal Transparent [✓ Wireframe]
NRRPLT-3102	ED: Text input for position and rotation Sandro implemented numeric fields in the Object inspector that enable the user to place the selected object at any desired position, and with any desired orientation.  robot Translate: 1, -1, 1.1 Rotate: -0.1, 0, -3.10111 [✓ Normal] Transparent Wireframe
NRRPLT-3123	ED: Object inspector Sandro implemented a widget in the 3D scene that pops up when one right-click on a 3D object. The inspector enables users to easily rotate and translate the object, but also to change its type of rendering.  robot Translate Rotate Normal Transparent Wireframe

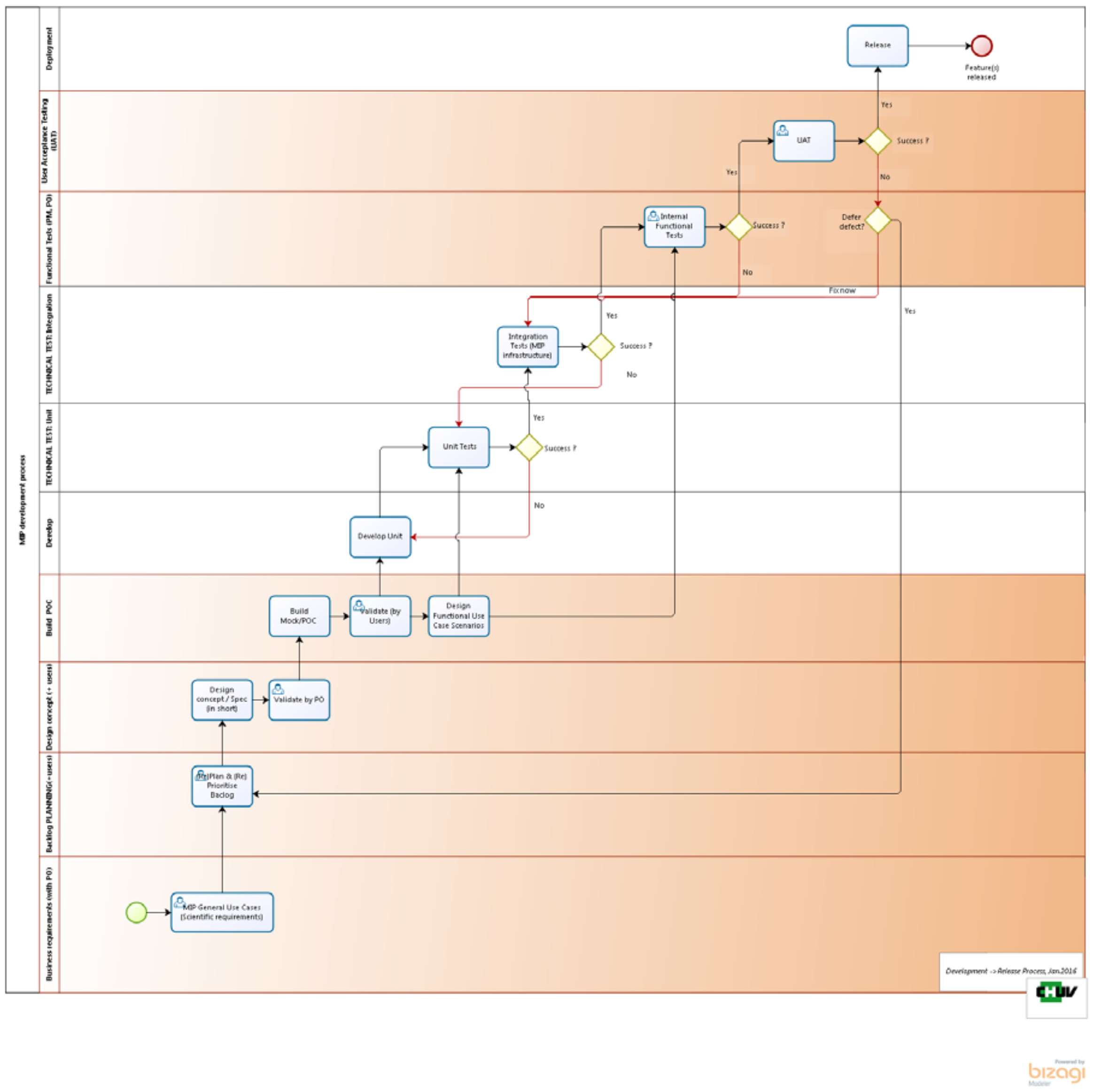
Subsequent steps

Once released, users can take advantage of the functionality which has been developed. If they have concerns or questions, these are routed through the standard "Feedback" mechanism on the site.

SP8 - Feature development process

SP8 has documented their current Feature development process, but user testing is currently focused primarily on internal users. For the sake of completeness, they've included parts of their development lifecycle with continuous integration testing and other non-UI testing activities.

User interaction is shown in this figure by the red background and little human icons in the process boxes:


MIP development process
Deployment
User Acceptance Testing (UAT)
Functional Tests (PM, PO)
TECHNICAL TEST: Integration
TECHNICAL TEST: Unit
Develop
Build POC
Design concept (+ users)
Backlog PLANNING(+users)
Business requirements (with PO)
Release
Feature(s) released
Yes
UAT
Success ?
No
Yes
Internal Functional Tests
Success ?
Defer defect?
No
Fix now
Yes
Yes
Integration Tests (MIP infrastructure)
Success ?
No
Yes
Unit Tests
Success ?
No
Develop Unit
Build Mock/POC
Validate (by Users)
Design Functional Use Case Scenarios
Design concept / Spec (in short)
Validate by PO
(Re)Plan & (Re) Prioritise Baclog
MIP General Use Cases (Scientific requirements)
Development -> Release Process, Jan.2016
bizagi

Annex F - Software Development Committee Charter

Software Development Committee

2016-08-29

Charter Working Group

Jeff Muller

Hans Ekkehard Plesser

Andrew Rowley

Andrew Davison

Colin McMurtrie

Anna Lührs

Thomas Heinis

Jean-Denis Courcol

Martina Schmalholz

Marc-Oliver Gewaltig

Bennie Weyers

Eric Müller

David Lester

Joan Gulley

Michael Thies

Colin McMurtrie

David Lester

(Others to be added on invitation of the Charter Working Group or SIB after the Charter is approved)

Purpose of the Software Development Committee (SDC)

The HBP Software Development Committee (SDC) shall address strategic cross-cutting software development issues in the HBP.

The Committee shall work independently to develop procedures, guidelines and standards to satisfy the strategic direction set by the HBP SIB. The SDC reports to the SIB through the Software Development Director.

The SDC will inform developers and administrators of the HBP platforms about procedures, guidelines and standards through workshops and living documents. It will coordinate its activities with the Data Management Committee and the Infrastructure Development Committee of the HBP.

The SDC will propose pragmatic approaches to ensure adoption of the developed standards, as unimplemented standards provide little gain to the HBP.

Mandate of the SDC

The SDC shall develop and communicate the following:

- HBP Software Engineering Guidelines
  - Common software development principles
  - Common development platforms
  - Source code control standards
  - Source code formatting standards
  - Documentation standards
  - Software practices for reproducible research
  - Recommendations for data handling in software products
  - UI standards
  - Protocol standards for public APIs and services
  - Data format recommendations
  - Software and source code licensing
- HBP-wide coordination of software development efforts in cases where multiple SPs work in the same or very similar fields
- Software security recommendations (in collaboration with Infrastructure Development Committee)
- Verification and validation standards (in collaboration with Infrastructure Development Committee)
- Guidelines for inclusion/adoption of existing projects
  - Evaluation criteria recommendations
  - Definition of a standard approach for interactions between HBP standards and standards already used in established community projects.
- Guidelines for Platform components
  - Definitions of classes of software
  - Support recommendations
  - Technology Readiness Level criteria (in collaboration with Infrastructure Development Committee)
- Awareness of software development activities across the HBP
- Recommendations on use of proprietary software
- Recommendations on shared infrastructure required to provide HBP software services
- HBP Software Catalogue
- Documentation of HBP Software Development Committee guidelines to address any relevant HBP milestones or deliverables.
  - For SGA1 this includes contributions to a number of milestones and deliverables, notably D11.2.2 – Software Engineering and Quality Assurance Approach

Annex G – Infrastructure Development Committee Charter

Infrastructure Development Committee

2016-01-17

Charter Working Group

Chair - Infrastructure Director, Karlheinz Meier

Deputy Chair - Technical Coordinator, Jeff Muller

SP5 – Jan Bjaalie, Timo Dickscheid

SP6 – Michele Migliore, Felix Schürmann, Eilif Muller

SP7 - Anna Lührs, Boris Orth, Colin McMurtrie, Cristian Mezzanotte

SP8 – Ferath Kherif, Ludovic Claude

SP9 – Andrew Davison, David Lester, Eric Mueller

SP10 – Marc-Oliver Gewaltig, Axel von Arnim, Luc Guyot

(Others to be added on invitation of the Charter Working Group or SIB after the Charter is approved)

Purpose of the Infrastructure Development Committee (IDC)

The HBP Infrastructure Development Committee (IDC) will address strategic infrastructure development activities which affect multiple Subprojects in the HBP.

The Committee will work independently in developing procedures, guidelines and standards to satisfy the strategic direction set by the HBP SIB. The IDC reports to the SIB through the Infrastructure Development Director.

The IDC will disseminate information to developers and system administrators of the HBP platforms with respect to these procedures, guidelines and standards by means of workshops and living documents. It will coordinate its activities with the Data Management Working Group and the Software Development Committee of the HBP.

The IDC will emphasise pragmatic approaches with a view to ensuring the adoption of HBP developed or external standards, since unimplemented standards are a hindrance rather than a gain to the HBP.

The IDC will collect and provide information about the standards of infrastructure components that are not owned by the HBP, but essential parts of the HBP infrastructure. It assesses the impact on HBP-internal standards, Service Level Agreements, support channels etc., and takes care that the externally defined standards are compatible with standards and procedures that are defined internally by HBP.

Mandate of the IDC

The IDC shall develop and communicate the following:

- HBP Infrastructure Engineering Guidelines
    - Common infrastructure architecture principles
    - Service configuration standards
    - Monitoring standards

- Security practices documentation
- System Documentation standards

- HBP Data Management Plan (“DMP”, M9 Deliverable and living document)
    - Harmonising the DMP with policy updates from the Data Governance Working Group (DGWG).
    - Recommendations and implementation for data handling in deployed services based on DGWG policies.
    - Implementation of data handling in deployed services based on DGWG policies.
- HBP-wide coordination of infrastructure development efforts in cases where multiple SPs work in the same or very similar fields
- Supports the SDC in developing software security recommendations
- Supports the SDC in developing verification and validation standards
- Production of standards recommendations for service deployment
    - Evaluation criteria recommendations
    - Definition of a standard approach for interactions between HBP standards and standards already used in established community projects.
    - Identify standard deployment methods for different environments (for example on HPC, on collab, with Docker, with regular devpi servers, etc.)
    - Provide recommendations (and later implementations) for services supporting HBP standard service deployment and monitoring.
- Guidelines for Platform components
    - Support recommendations (where 3rd party components are deployed with minimal changes)
    - Technology Readiness Level criteria (in collaboration with Software Development Committee)
- Reporting of all service updates to the official service catalogs and databases.
- Recommendations on shared infrastructure required to provide HBP software services
- Documentation of HBP Infrastructure Development Committee guidelines to address any relevant HBP milestones or deliverables.
    - For SGA1 this includes contributions to a number of milestones and deliverables, notably D11.3.2 – Data Management Plan (M9), but also including other deliverables in other SPs.

Annex H: References

[1]Beedle M *et al*. (2001). *Manifesto for Agile software development*. Available (viewed 2017-11-27) at: http://agilemanifesto.org/

[2]Anderson DJ (2010). *Kanban: Successful evolutionary change for your technology business*. Sequim, WA: Blue Hole Press. 262 pp.

[3] *HBPMedical/mip-microservices-infrastructure*. Available (viewed 2017-11-27) at: https://github.com/HBPMedical/mip-microservices-infrastructure

[4]*Status Cake*. Available (viewed 2017-11-27) at: https://www.statuscake.com/

[5]Exoscale. *Runstatus*. Available (viewed 2017-11-27) at: https://www.exoscale.ch/runstatus/

[6] *HBP SP8 Repository: Development process*. Available (viewed 2017-11-27) at: https://hbpmedical.github.io/development-process/

[7] *HBP SP8 Repository: Software catalog*. Available (viewed 2017-11-27) at: https://hbpmedical.github.io/software-catalog/

[8] SP7 D7.7.5. *High Performance Computing Platform* v1. Available (viewed 2017-06-29) at: https://sos.exo.io/public-website-production/filer_public/42/2f/422fb62e-c232-4b72-bfc7-9dd6b23f6578/d775_rup_m30_accepted_20160803.pdf

[9]Heroux M, Ross Bartlett R, Willenbring J (2012). *Software engineering principles: The TriBITS lifecycle model*. Available (viewed 2017-11-27) at: http://www.sandia.gov/~maherou/docs/HerouxTribitsOverview.pdf

[10]Available (viewed 2017-06-29) at:

https://www.nasa.gov/sites/default/files/files/39_Agile_Estimating_for_NASA_CAS_2014_Tagged.pdf

[11]*NEST: Developer space*. Available (viewed 2017-11-27) at: https://nest.github.io/nest-simulator/

[12]*nest/nest simulator*. Available (viewed 2017-11-27) at: https://travis-ci.org/nest/nest-simulator

[13]*The NEST simulator*. Available (viewed 2017-11-27) at: https://github.com/nest/nest-simulator

[14] *Human Brain Project forum*. Available (viewed 2017-11-27) at: https://forum.humanbrainproject.eu

[15]Annex G: Technology readiness levels (TRL) of *Horizon 2020 – Work programme 2016–2017*. Available (viewed 2017-11-27) at: http://ec.europa.eu/research/participants/data/ref/h2020/other/wp/2016_2017/annexes/h2020-wp1617-annex-g-trl_en.pdf