Stage-2 Proposal with YARR

This proposal was only possible thanks to the work of many people who developed concepts, code and ideas throughout the years.
The speaker has tried to summarize what is the logical way forward and takes full responsibility for any inaccuracies and mistakes in this presentation.

A huge thanks to: Elise Le Boulicaut, Bruce Gallop, Timon Heim, Maria Mironova, Angira Rastogi, Zhengcheng Tao, Alex Toldaiev and many more for their work and ideas

Introduction / Outline

- Describe how YARR can be integrated into the full ITk-DAQ required infrastructure as described in the TDAQ specification and requirements (including the referenced ITk requirements) documents.
  - Highlight the flexibility and benefits of such an approach
- Describe a development model and timeline that adheres to the guidelines provided by the ITK online software management team (ITk week introduction)

- Certainly not at the level of a complete project description, but aiming for enough information to assess the feasibility and advantages of this approach.

YARR, quick recap

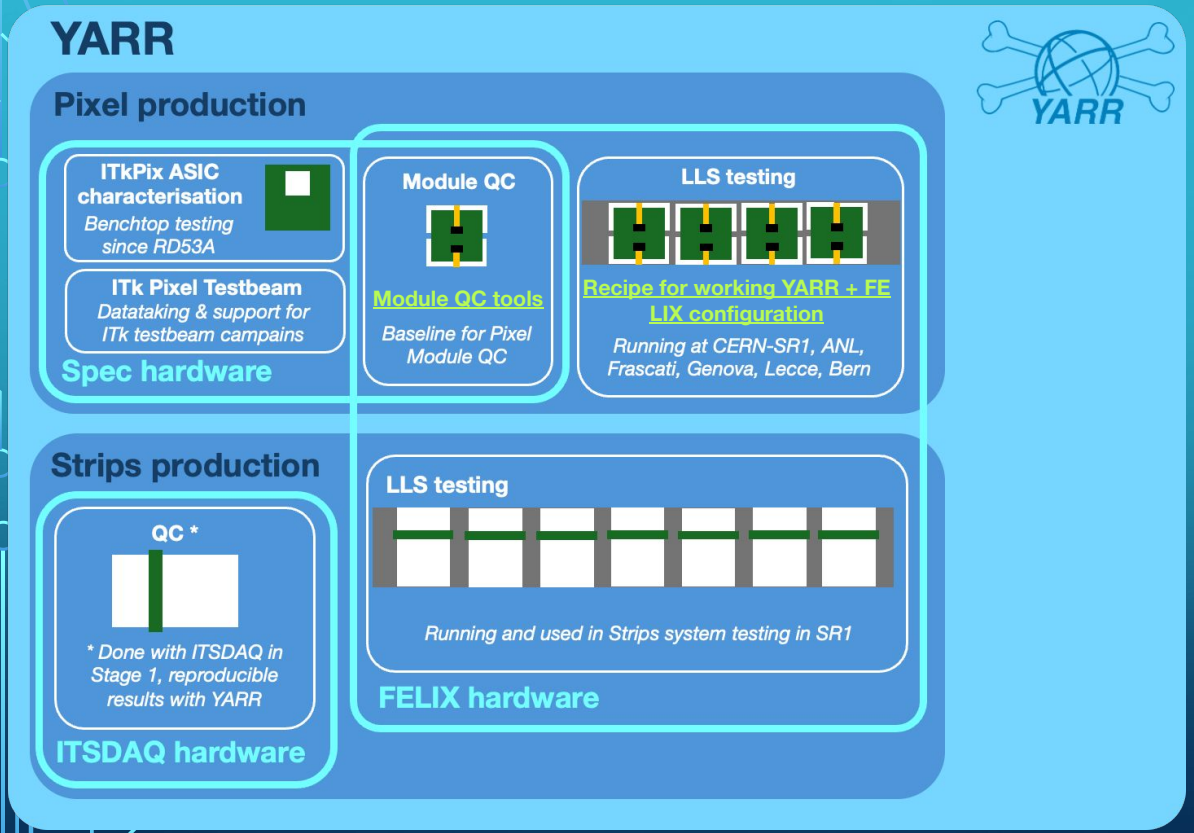
YARR is a (ITk-oriented) readout system based around the concept of moving *intelligence** from the FPGA firmware into the host computer software.

- Follows the same base concept as the phase-II DAQ architecture and setup

Openly developed in gitlab

- Modular by design: conceptual pieces separated in individual libraries
- Addresses and takes advantage of synergies between Strips/Pixel projects
- Collaborative and modern software development with reviews / components / CI / etc..
- Regular releases (latest: 1.5.1); software documentation (new ITk pages coming soon)

YARR, quick recap



Heavily used and developed for ITk use-cases

Baseline for Pixel module QC tests in lab settings

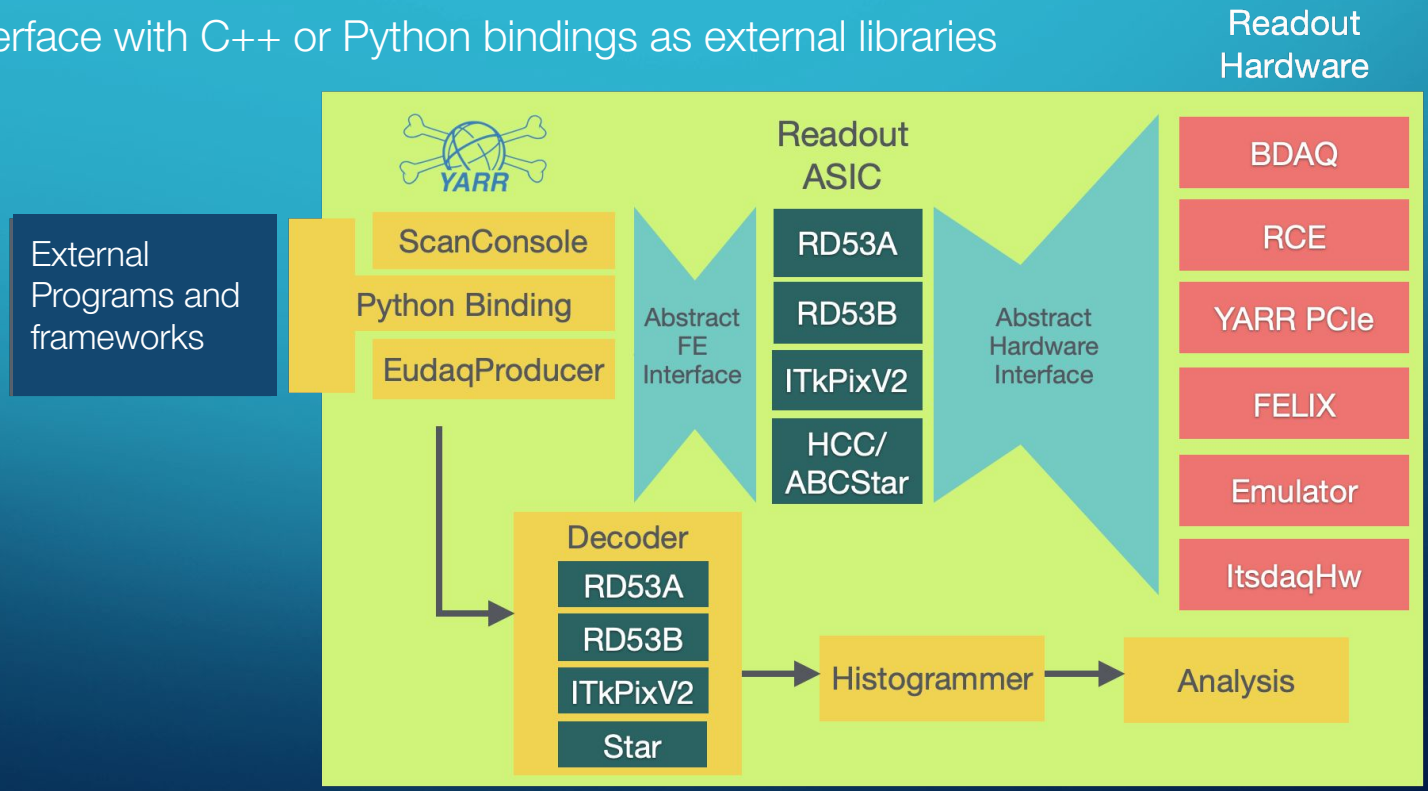
Baseline for stage-1: interfaces with FELIX

- Functionalities and performance developing as we gain experience in this configuration
- Successfully tested in many integration sites: CERN-SR1, ANL, Frascati, Genova, Lecce, Bern (see e.g. here for a recent summary and here for step-by-step instructions)

YARR “building” blocks

Abstract functionalities, efficient reuse data-flow logic for Pixel and Strips

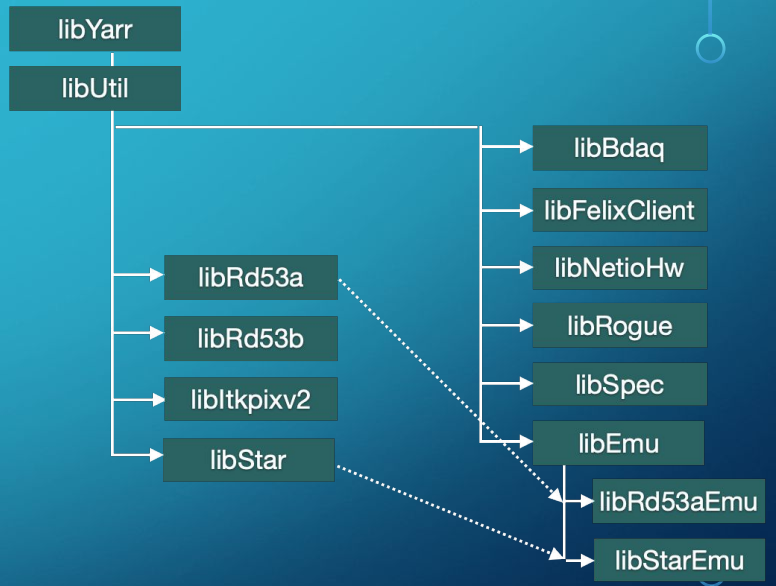
- Interface with C++ or Python bindings as external libraries



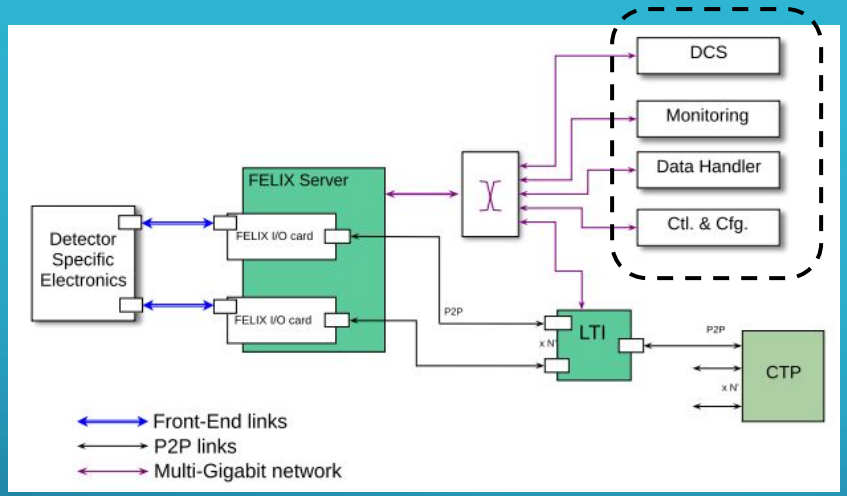
YARR modularity

- Software organized in libraries with well-defined scope, functionality and interfaces
- Flexible cmake structure allows to only build necessary components for the specific application
  - Minimal cross-library dependencies
  - Minimal external dependencies pulled only if needed

- **All** functionalities can be used in two main ways
  - as a (set of) libraries
    => preferred for phase-II DAQ software
  - as a standalone readout software through a minimal CLI
    => easier for lab testing e.g. individual module QC



Phase-II Readout ecosystem



- Simplified view, see Nikolina's slides for a more complete picture
- ITk-specific code interfaces with software / interfaces provided by the underlying architecture

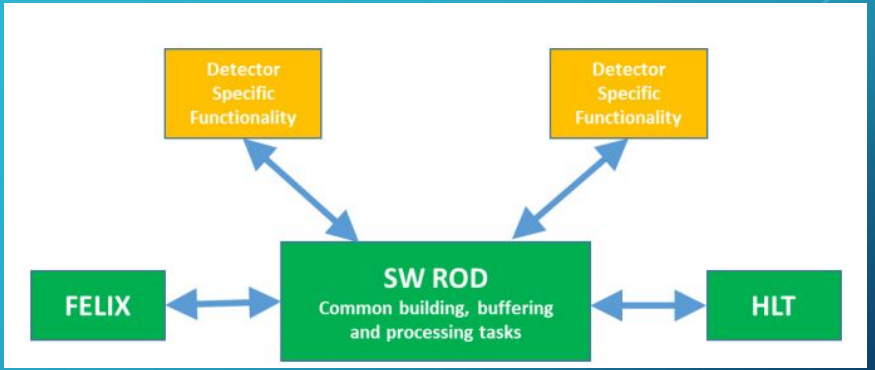
We envision ITk-specific software to be developed for each major piece that heavily uses YARR libraries across projects (DH, monitoring, Ctrl/Cfg, DCS, …) to maximize code reusability, ensure harmonization and addressing especially the pieces that are most FE-specific.

Data Handler: ITk-specific Software

Data Handler will be configured differently in

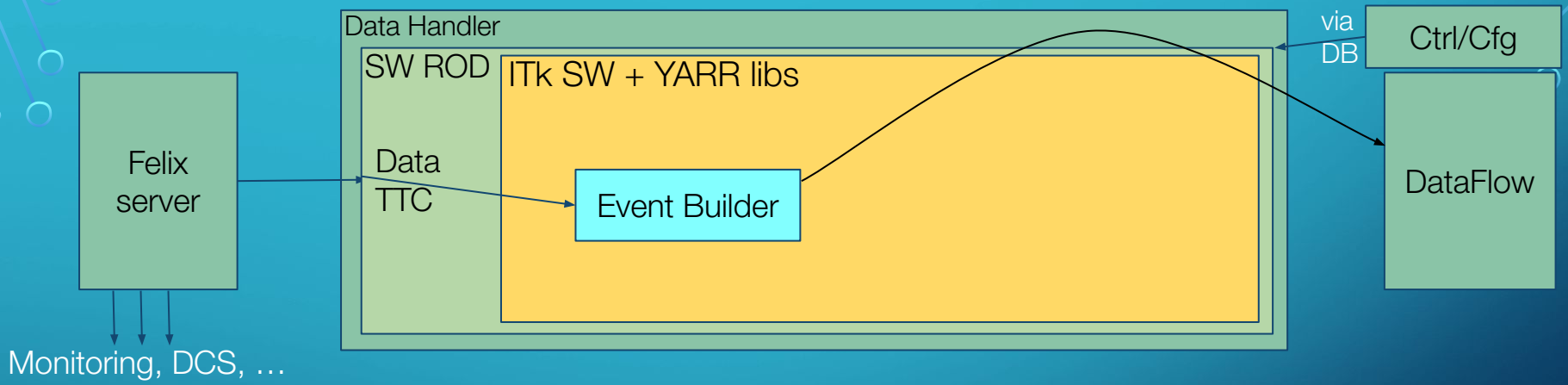
- Data-taking mode
- Calibration mode

As stated in the specification documents.



While the configuration differs, lots (most) of the overall code should be shared to maximize reusability and minimize problems and maintenance.

DH: Data-taking mode

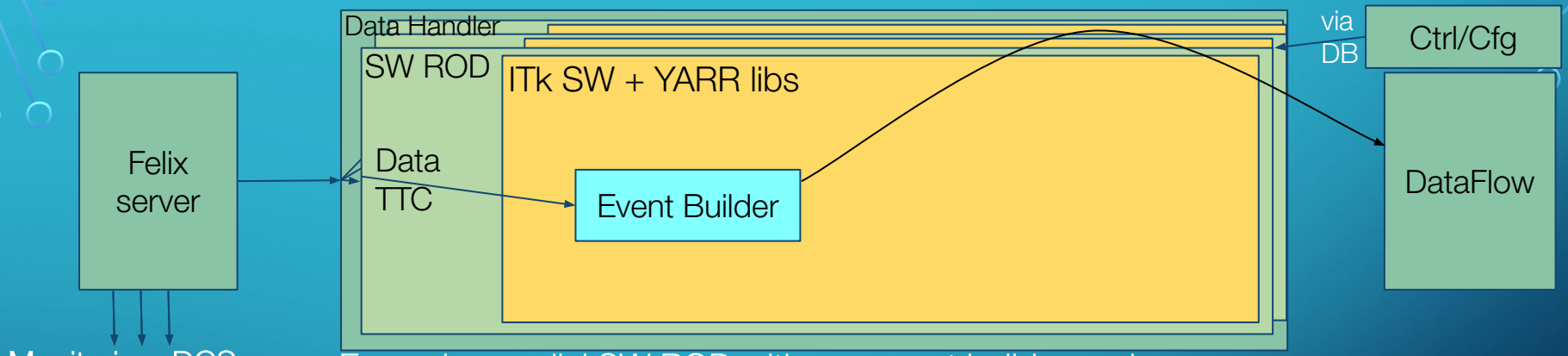


Simplest implementation (not robust) sees a simple plug-in building the event using minimal detector-dependent information.

State-aware application adhering to ATLAS RC FSM.

Interface with config DB / OKS via provided tools.

DH: Data-taking mode



Example: parallel SW ROD with one event builder each
Alternative: parallel Event Building threads from one SW-ROD instance

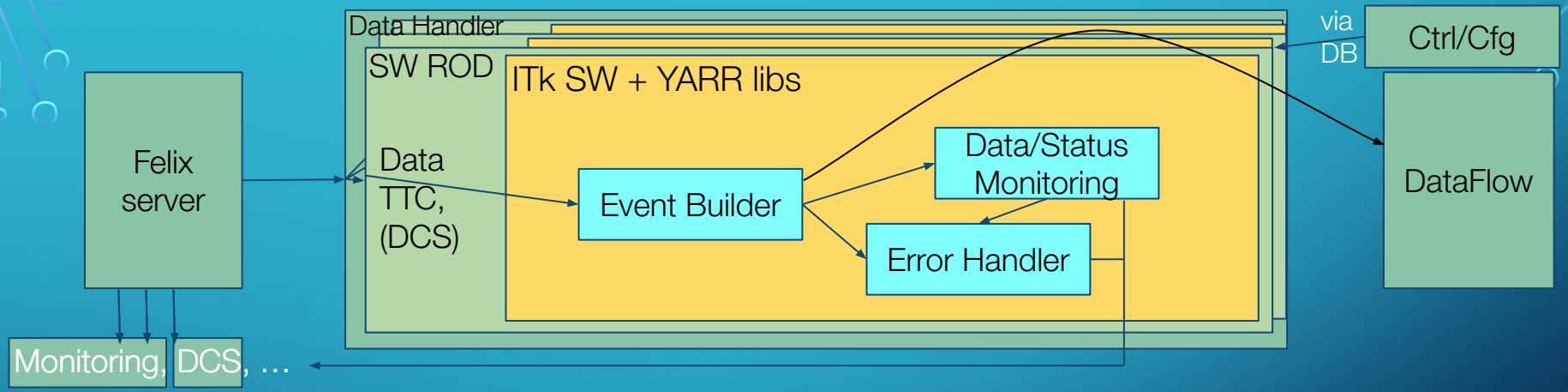
Scaling having multiple instances running in parallel

- Can be achieved either at SW-ROD level or at the ITk Event Builder level, as preferred
- For some features (see next) some knowledge of “structures” (i.e. modules) is needed

Control over multi-process through TDAQ IPC tools

This is independent of codebase, choice should be evaluated and discussed

DH: Data-taking mode, monitoring

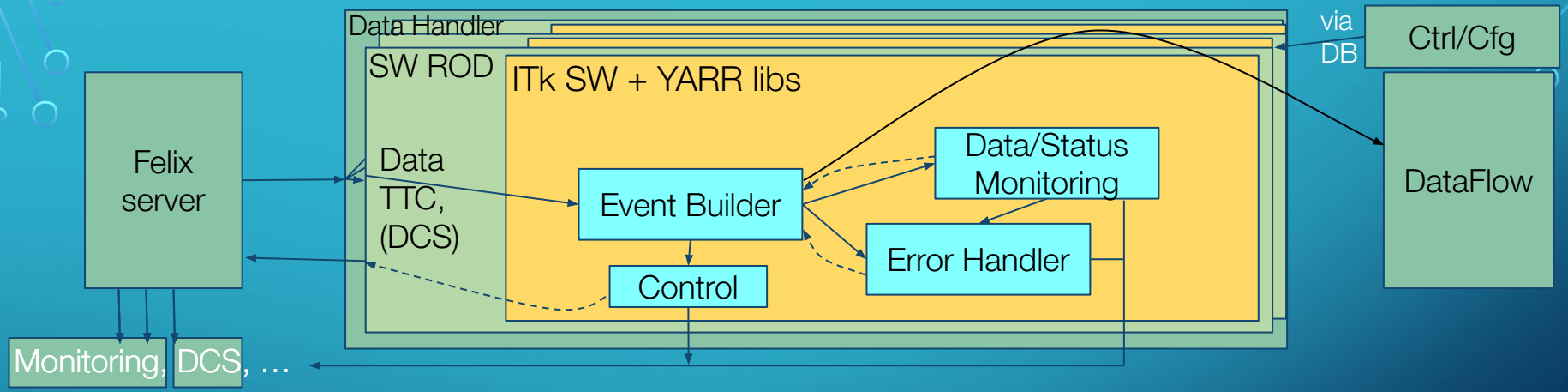


For fast error detection and recovery cycles, need minimal error checking and monitoring from within DH (as expected from specification/requirements)

- For specific fast-error detection, some limited DCS information is needed and useful to read out directly from Felix server (e.g. strips heartbeat, …)

Data for comprehensive monitoring pushed to dedicated servers.

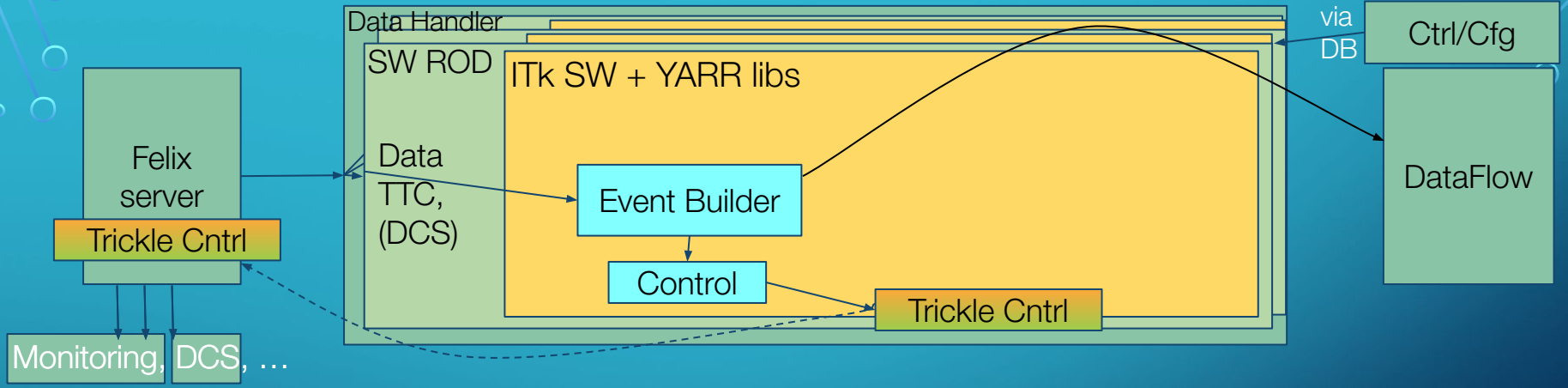
DH: Data-taking mode, monitoring



Fast recovery **might** need a back-communication channel (as allowed from spec), the need for that should be evaluated and discussed with TDAQ, independent of codebase and driven by expectation of operational reliability and response times.

- If only slow response needed, for instance, can go through Monitoring/Ctrl&Cfg
- Experience from phase-II system testing and Run 1-3 operation

DH: Data-taking mode, trickle config



Continuous configuration of FE chips (Strips and Pixels) **needed** and needs to be supported instructing Felix what data should be sent and update it when necessary (as instructed by the Ctrl/Cfg servers) directly in Felix server or in DH, and either way requires more experience, a dedicated discussion with TDAQ, and can't be settle unilaterally.

Monitoring, Ctrl&Conf

Monitoring

Most of heavy monitoring information pushed to a dedicated server that include specific ITk software

- re-use as much as possible common YARR library histogramming/analysis tools when FE-specific knowledge is required to ensure coherence and portability

Control & Configuration

Needs ITK-specific software that can be developed in parallel, well-defined interface.

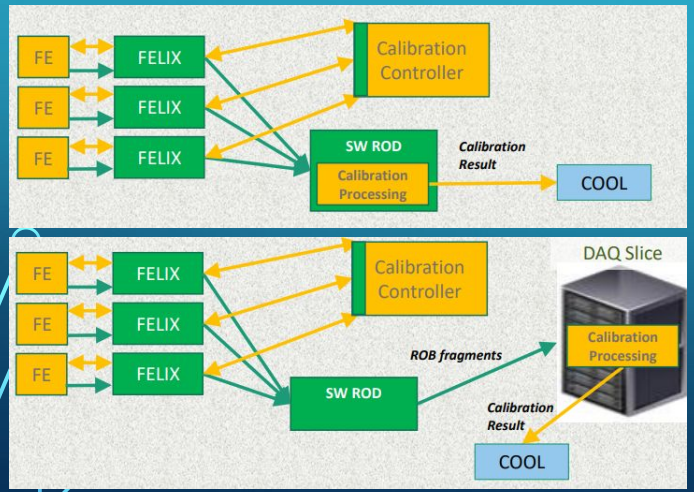
- Can and should re-use as much as possible common YARR library tools when FE-specific knowledge is required to ensure coherence and portability

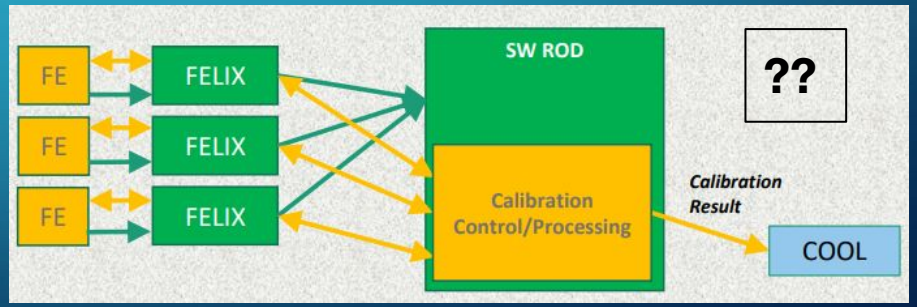
The devil is in the details. For both, need to identify a variety of non-local (FE) problems and take appropriate actions consistently.

DH: Calibration mode

DH can be reconfigured for calibration purposes.

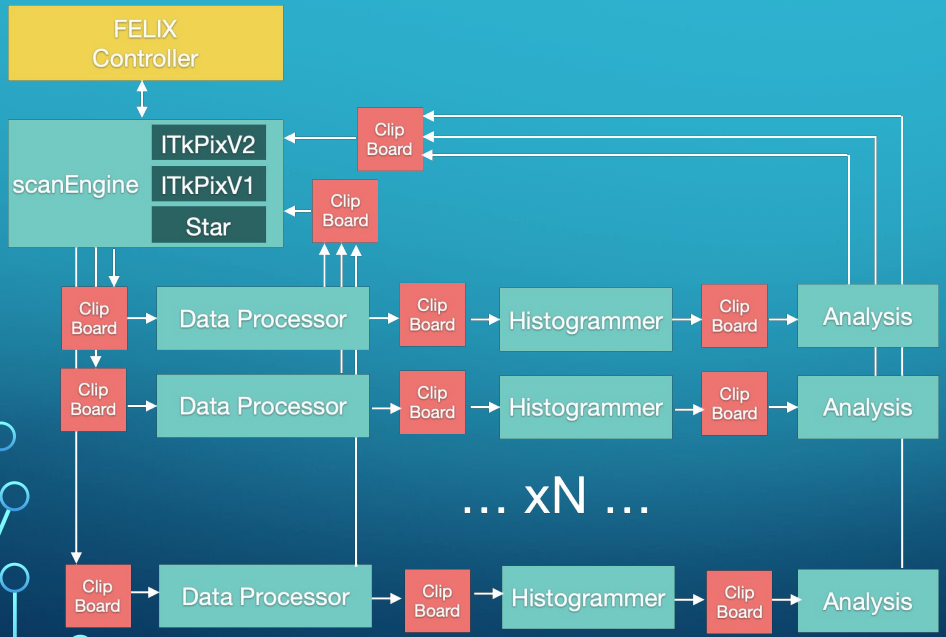
- Several paths envisaged, but for ITk there's a need for a fast feedback mechanism, as supported by specifications/requirements (e.g. Req. 2.22)
- most straightforward would require a direct two-ways communication between DH ITk code and Felix server.





DH: Calibration mode

YARR provides several utilities to perform calibrations that have been refined through experience.



While conceptual flow of calibration procedure prefers the more localized solution to allow fast feedback, the code actually allows either.

Processor separation highly configurable, pipelined, and naturally scalable.

Can interface to appropriately configured SW-ROD as shown earlier.

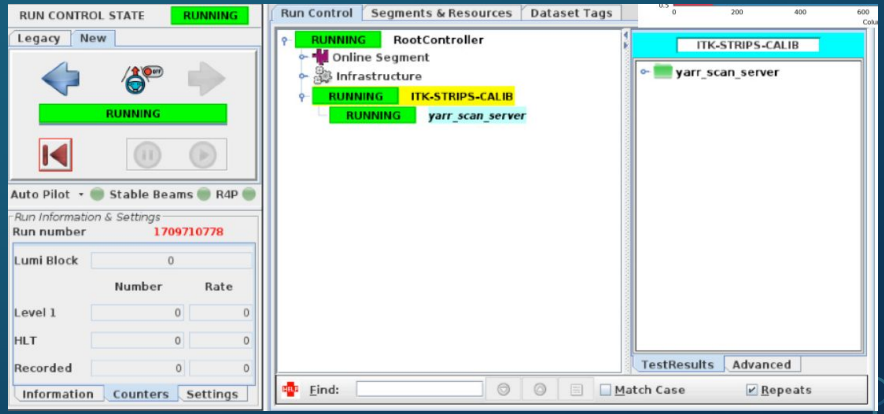
Choice driven by performance rather than codebase.

Clip Board → Can also be network interface

Practical integration in TDAQ software suite

Proof-of-principle developed to show how ITK-dedicated software interfaces easily with YARR within the full TDAQ software suite as for other TDAQ components

- Compile within TDAQ software
- FSM machine simple implementation
- Configuration via DB (OKS schema)
- More information in Zhengcheng's slides[1,2], git repository



All of this maintaining the ability to use as a standalone utility when needed for other usages.

Usage of TDAQ tools

We see no showstopper in fully integrating TDAQ tools

CMake: see also previous slide

Logging: integrate/wrap/switch from current logging simple framework (spdlog)

Inter-Process Communication & Messaging: used primarily in the ITk-specific pieces that themselves interface with YARR libs

CMake (prev slide), logging, inter-process comm, run/config services (prev slide), …

Configuration Services, GUI, … also envisaged to be developed largely in parallel

Aside: Interfaces evolution

YARR software can interface with no issues with existing *TDAQ* architecture.

Nevertheless, there are examples of concrete use-cases where we might want to change how things are organized currently.

Example: direct FE communication for fast reaction presented earlier

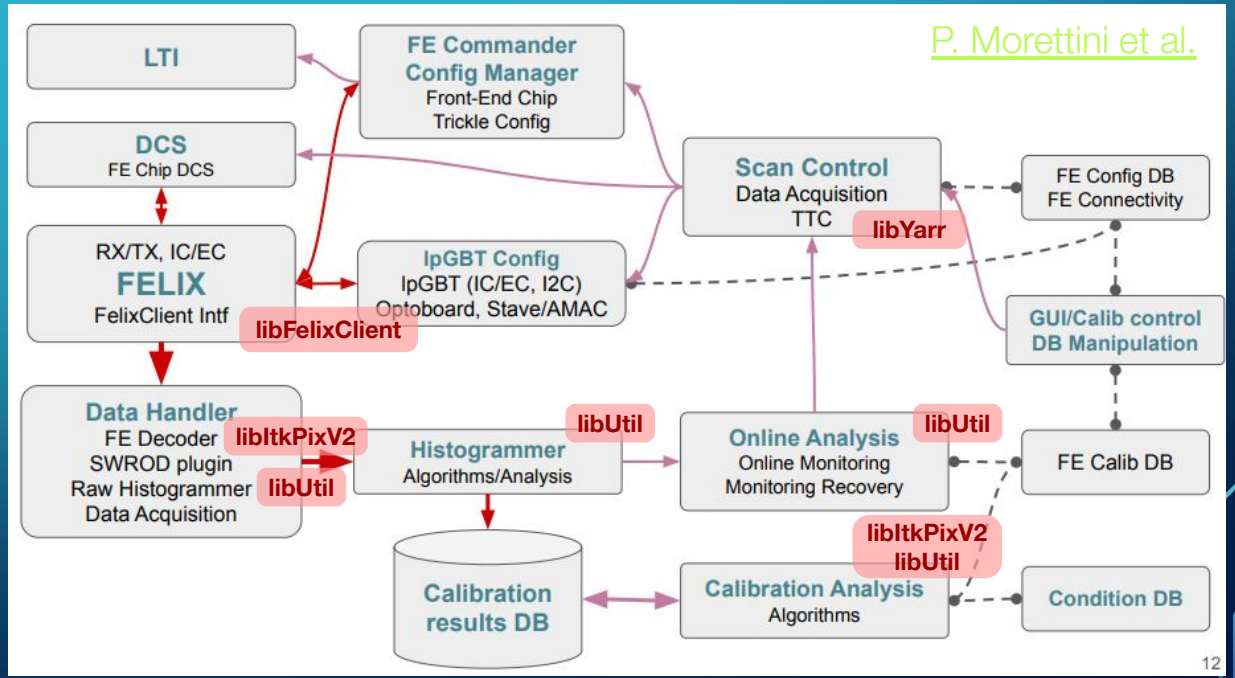
- **Independent of codebase**, just showing an example of how we can't consider these interfaces fixed at this stage, as we gain more experience and have a more operation-oriented approach

Aside: comparison with prev. shown block diagram

Many of the components shown before that are ITk-specific can be mapped to existing YARR libraries (FE-specific knowledge) and new algorithms that can reside in dedicated repos using those functionalities (see also next slide).

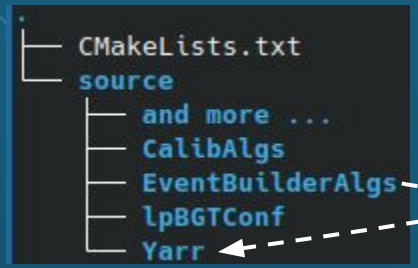
Also
- Emulation [libEmu, …]

Avoid code duplication re-using functionalities through YARR libs and implement only the dedicated logic.



Software organization

Given the discussions and inputs received, we tried to envisage a structure that would maximize the benefits of various ideas and arguments people brought up.



Each folder in source can also be in a separate git repository

YARR libs enter as one folder and the various algorithms can use its functionalities freely while implementing a lightweight logic and specific error handling/messaging/etc..

Working environment allows to work in parallel on individual projects.

Separate git packages (submodule or via cmake) for development of components whose interfaces are not expected to evolve drastically and are more atlas-operations-specific. Link those to YARR libs for reuse code and avoid versioning nightmares.

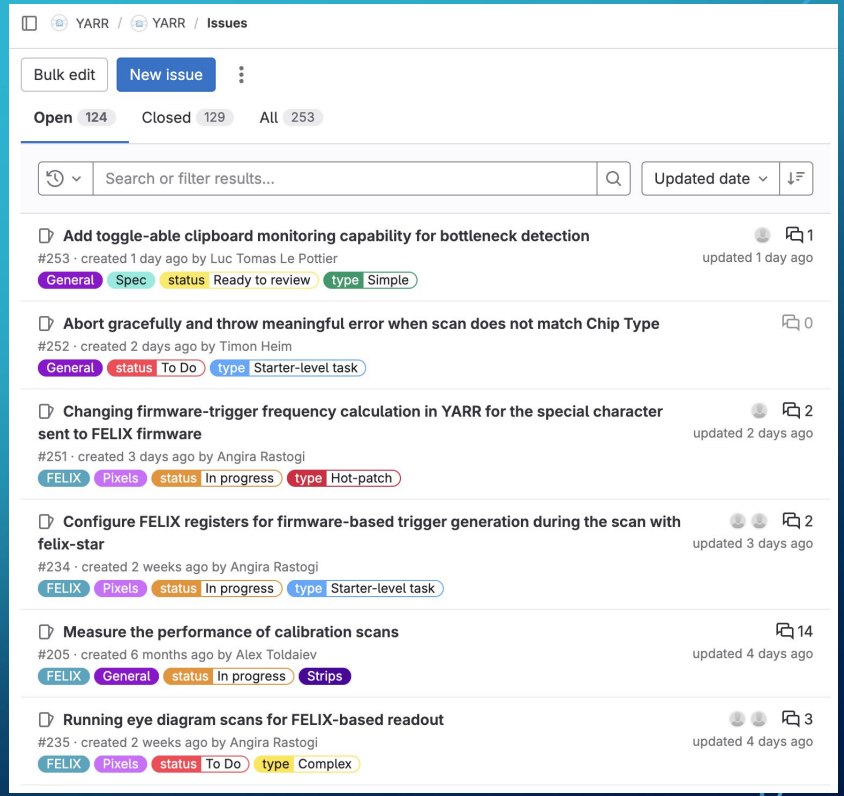
Expected need to evolve interfaces and tight connections between components within YARR libs strongly favor keeping those as a single repository.

Software Development Model

- Clear need to agree on software development model, to define how we intend to develop the existing codebase as something suitable detector operation
- Presented proposed workflow at ITk SW meeting Dec 7
  - Goal was to openly outline the organization of development and practices to which every developer converges for constructive cooperation long-term
  - Also provide an entry point for new developers and gives them a detailed guideline how they can help with the development
- Outlined best practice workflow through git issues and MRs, and how different kinds of developments (bug-fix, simple MR, architectural changes) could happen

Software Development Model Implementation

- Started implementation of aspects of the development model in YARR repo
- Have been very actively using git issues to keep track of developments
- Implemented **labels** to classify issues, based on
  - ASIC, Readout hardware, Processors
  - Status and complexity
- In the process of introducing the concept of **codeowners**, to distribute also review process better
- Currently updating contributing guidelines to be more suitable for a larger number of new developers
- Clearly iterative process → aspirational and ongoing deployment, feedback welcome as usual!



Documentation

- **User documentation:**
  - Existing tried and tested YARR documentation here: https://yarr.web.cern.ch/yarr/
  - Additionally dedicated user documentation for different use cases, e.g. YARR+FELIX or module QC
  - Working towards providing higher level documentation/landing page in official ITk docs
- Established successful channels for **user support** → dedicated mattermost channels (ITk Online SW, YARR) and git issues
- **Architectural documentation:**
  - Plan to have Architectural Design Records (see Slide 20 of SDM) → record for why certain decisions were made, as well as documentation for new developers
    - Log of these changes comes automatically through git within the same repository (yet another reason to keep e.g. YARR components in a single repository)

Engagement

- **Successful code development requires community engagement**

→ aim to build a community that is actively developing software towards a common goal

- Progression of new developers can look as follows:
  - Good documentation allows for efficient training of new developers in existing code (user documentation & architecture design records)
  - First contributions through well documented starter-level issues
  - With experience, contributions to MR & code review, as well as longer-term developments
  - Engagement is not just delivering code, also presenting it → e.g. through status updates at ITk/Upgrade week
- **Welcome contributions on all levels** (ranging from newcomers to experts) through the varying levels of MRs and code reviews
- Project structure ensures parallel developments will be possible
- Coordination and early communication of developments are crucial, along with tracking of ongoing tasks and developments

Organization: same blocks, different focus

ITk Online SW Coordination
Coordinate all activities and priorities, ensures overall project stays on track and it's developed according to specs and common philosophy.

Schedule / Requirements
Design and then keep track and update schedule and requirements

Pixels
FE-specific interaction.
DH algorithms development (data-taking, config, ..). Emulation

Strips (?)
FE-specific interaction.
DH algorithms development (data-taking, config, ..). Emulation.

DB
Interfaces with DB components.

FELIX Firmware
ITk point of contact for firmware development and features request organization.

HW Interfaces
Should ramp down for Stage-II, but still needs support in the medium/long term in case it's used directly for calibrations.

Core Software
Base architecture evolution, close watch to interaction among pieces.
CI development. Maintenance.

User Interfaces
Development of dedicated GUIs and user-interaction utilities, including for integration / commissioning.

SW performance / testing
Benchmarking, development and and common detailed tests analysis.

TDAQ interface
TDAQ liaison and development of tools that are not pixel/strips-specific.
Co-coordinate development of DH algorithms with Pixel/Strips.

Organizationally, while the current structure can remain rather similar and evolve into Stage-II, practically the type of development and weight on each box will shift.
Also, what was "YARR" now denotes a broader "ITk SW + YARR" development area

Organization: high-level

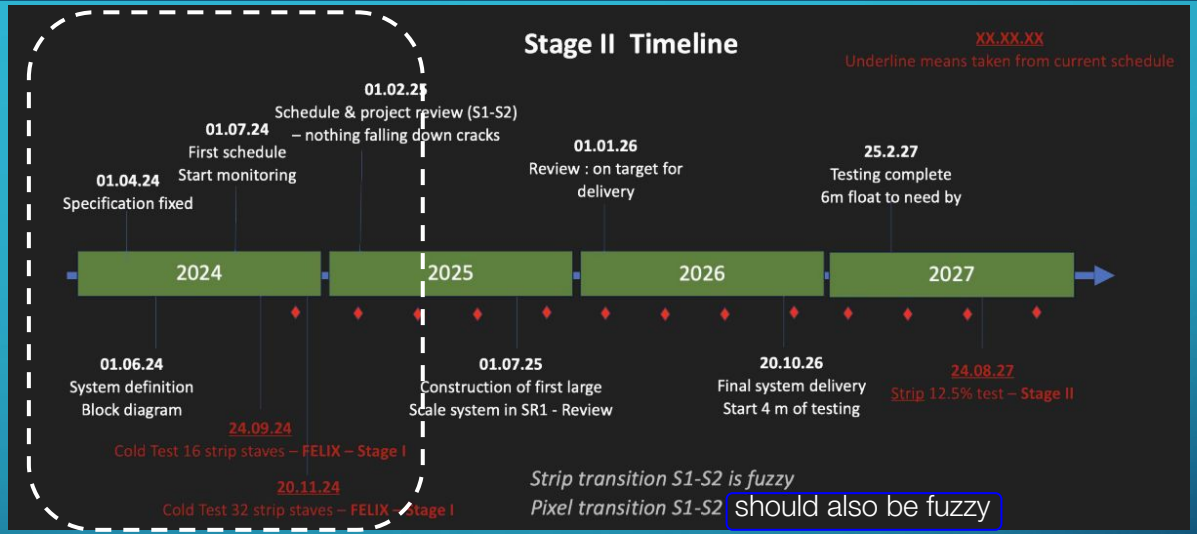
- Big milestones should be set with the overall schedule (next slides)
- At regular intervals, define tasks that enter the next milestone
  - Prepared by coordinators, discussed in overall meeting, gather feedback
  - Tasks organized as gitlab issues with appropriate labels
  - Ensure wide variety of tasks that suit various levels of expertise
- Aspire to engage everyone in both development and maintenance
  - bug-fixes, user support, …
- Dedicated “sub-groups” meeting for very technical issues that are localized, while discuss in the overall meetings cross-box decisions and architectural changes

Connection to Stage-1 developments

- We think stage-1 is far from “done” and we will continue to learn from it, therefore it’s important to have a system that enables us to easily transfer knowledge between the two
  - Stage-1 tests can and should inform stage-2 evolution, recent strips scalability tests
  - Most developments are indeed contributing at the same time to stage-1 and stage-2, should be considered a single set of developers
- Organizationally, we shift focus but maintain a structure that allows the experience and information gained to not be lost in “other meetings”
- Existing YARR code is a huge boost in avoiding repeating problems and re-learning lessons throughout the years; at the same time the current structure allows a clean and self-contained direct development towards the final system

Timeline

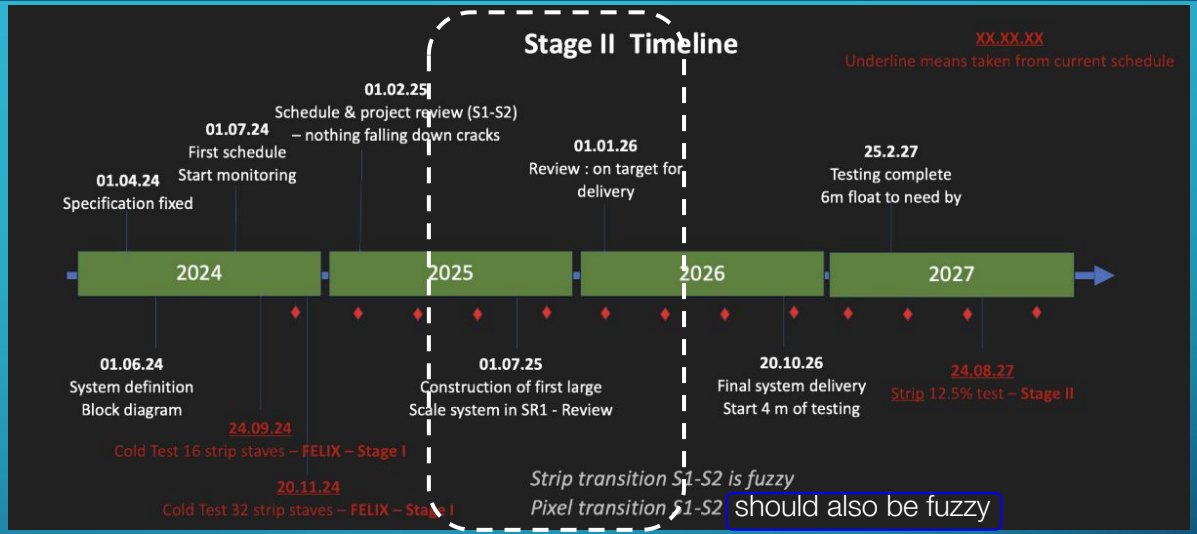
As presented at the last ITk week



Short-term (in parallel), leading to (1) experience of a scalable system, (2) experience in structuring code as suggested

- Robustness and features implementation of Stage-1 system, prepare/learn/evolve from LLS and dedicated tests with multiple modules [6mo – 1yr]
- Detailed scalability analysis with small lab systems for insights in medium-term [6 mo]
- Setup of repository and development structure for non-YARR components [1-2 mo]
- Start development of initial logic for event building [6 mo] and other components [1 yr]
- Integrate other TDAQ tools (e.g. logging, monitoring, …) in the framework to gain experience on best practices [1 yr]
- Create and discuss more detailed schedule [2 mo]
- Discuss ITk-TDAQ interfaces and advantages/compromises with an eye to operations [6 mo]

Timeline



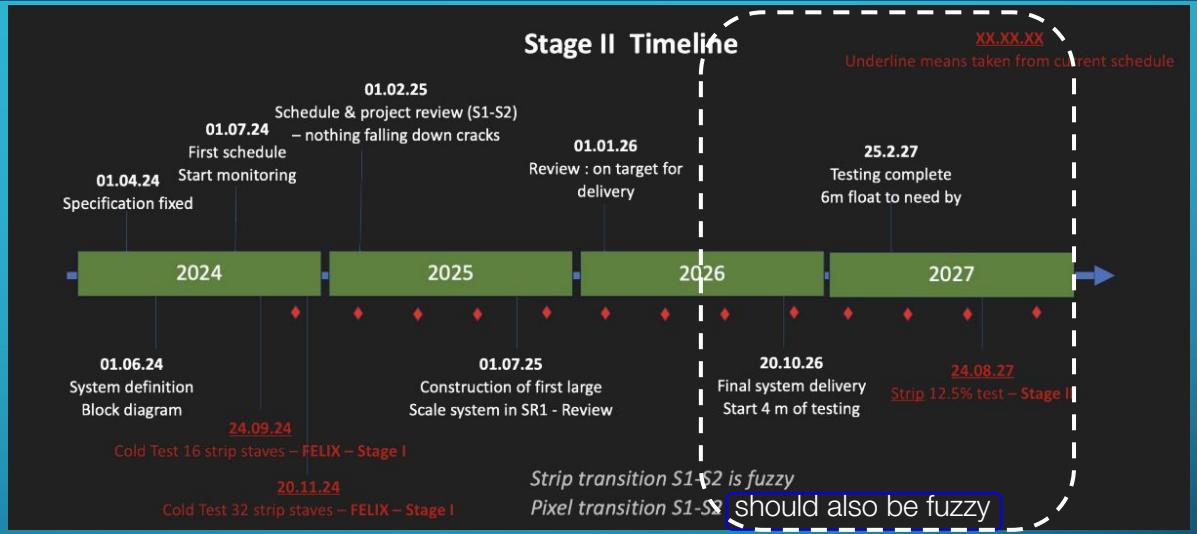
Medium-term, leading to (1) large system tests [mid '25], (2) experience in full system core components

- Refine infrastructure to bring all pieces together correctly, confirming or (likely) analysis of scalability performance
- Any evolution of YARR libs organization will likely happen at this stage, based on experience gained so far

- First full implementation of all TDAQ-interfacing pieces
- Full Implementation of trickle config in its final form (discussion starting in short term, incl. components tests)
- Gain experience in small and large tests

- Continue strong support of QC and LLS sites that will be fully using and stress-testing the systems' components
- Incorporation of lessons from stage-1 tests is automatic due to shared YARR code in libs

Timeline



Long-term, leading to system delivery for large-scale tests and deployment

- Refine and deploy a detailed analysis of performance based on a full system test
  - Not a binary “work/not-work”!
- Keep improving various components based on medium-term tests results
- Prepare utilities specific to integration and commissioning

Timeline

Overall not yet a full schedule , but wanted to highlight a few key concepts

- Stage 1-2 transition is not binary and the two will need to co-exist and learn from each other; this proposal minimizes the burdens for this to happen with minimal person-power effort
- The Strips project is fairly ahead in their needs and see no show-stopper for TDAQ implementation following something I believe being fully compatible with this proposal.
- Explore options (and understand their implications) early, but make decisions based on experience and tests (e.g. re. communication in DH): can expect interfaces to have to change, but need to be well motivated and tracked.
- Lots of development can happen in parallel, but we always will need core developer to support users and help improving the system, beyond designing proof-of-principle code
- Design tools and uniformity of code to ensure trained experts can work and contribute to integration, commissioning and know the system during operations

Advantages of using YARR as part of the codebase

While we highlighted several points throughout the presentation, there are a few more important long-term considerations

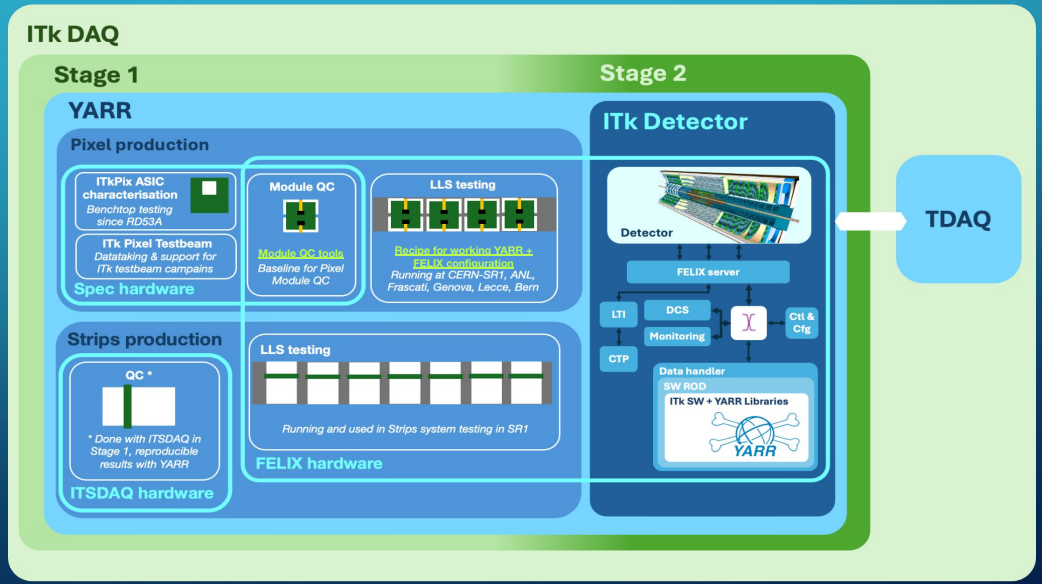
- Usage of YARR all the way from module QC, through LLS testing, integration and operation ensures:
  - Comparable testing results all throughout the different detector construction stages
  - Transfer of knowledge and experience between the different stages
  - Training pipeline for shifters/experts for ITk operation during HL-LHC

    → e.g. significantly lower ramp-up needed for shifters who have worked on ITk production if they can use the same core software during operation
- Ensure long-time availability of experts
  - Future R&D will involve readout of new chips, testing, etc.. having a portable system as YARR that can support those ensure that who works on that is also trained to easily understand and work on core HL-LHC ITk DAQ readout (we know we will have lack of expertise during operations
  - Common core codebase for Strips and Pixels enlarges the pool of available experts on this code

Conclusions - I

Outlined how a stage-II project that incorporated YARR can be achieved

- YARR should be seen as a set of libraries with detailed knowledge of the specific FE and its “quirks” while several other components need to enter the final ITk DAQ software
- fully compatible with usage in the final ITk DAQ online software
- Highlighted some key architectural points that might need further discussion, to explicitly decouple the codebase discussion and choices that any codebase will have to make



Conclusions - II

Overall incorporating YARR as work-horse in ITk DAQ software provides many advantages:

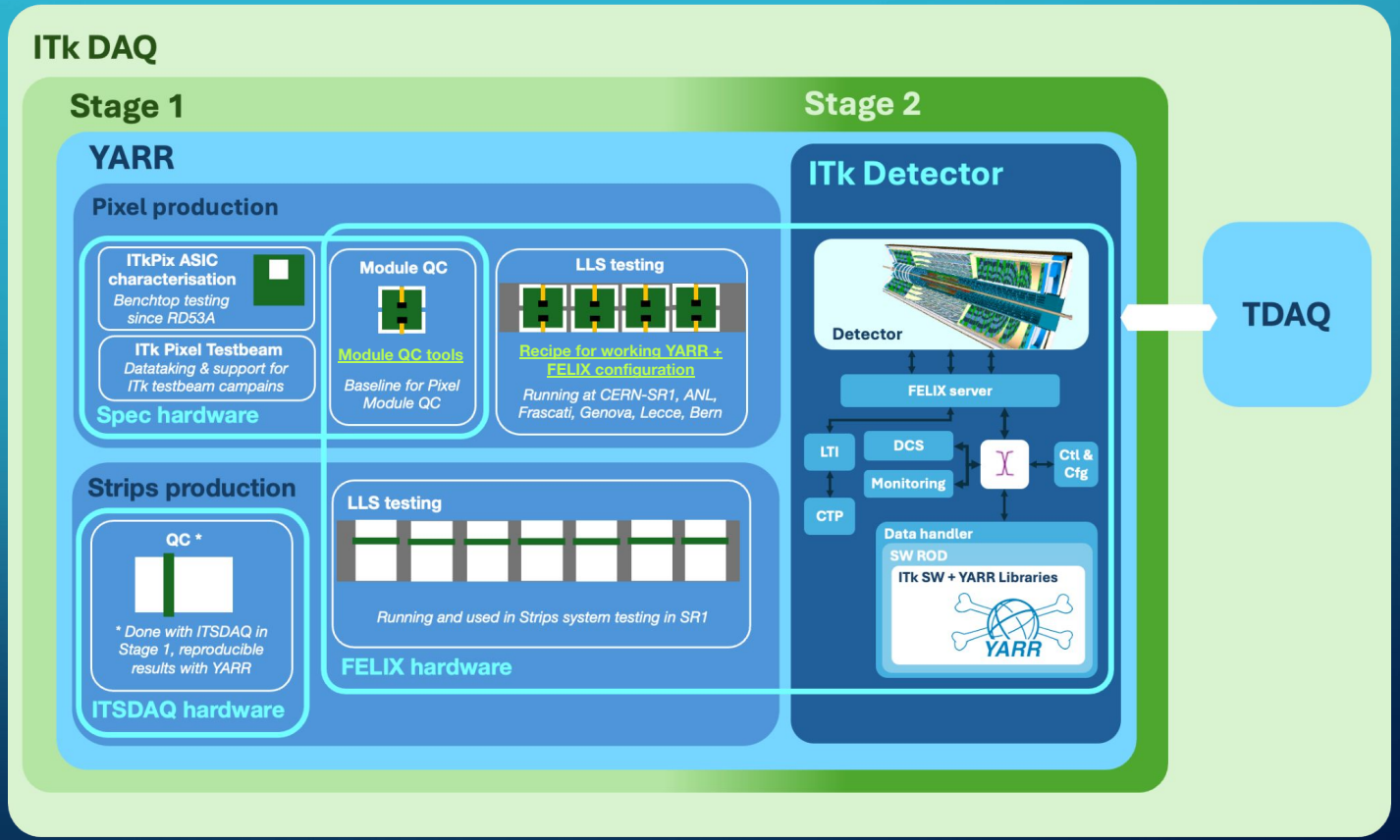
- Many years of experience and robust collaborative development (gitlab MR, regular releases, …), yet kept evolving to meet the needs and in view of the experience gained while providing a stable software
- Ensures continuity with pixel QC, stage-1 developments and allows us to continue to transfer the knowledge as we go forward
- Exploits at best synergies between Strips and Pixel developments
- Highest chances to keep forming experts that fulfill the needs also during HL-LHC operation

Presented a robust software development model that ensures good code quality, yet allowing co-existintence of short and long-term developments and taking maximum advantage of their synergies.

Started to sketch an organization and timeline for the project that is compatible with ATLAS needs.

We all work best when we are focused on solving problems using the same codebase, several interesting ideas and concepts are now spread out and we hope that can be unified going forward.

BACKUP


ITk DAQ
Stage 1
Stage 2
YARR
Pixel production
ITkPix ASIC characterisation
Benchtop testing since RD53A
ITk Pixel Testbeam
Datataking & support for ITk testbeam campains
Spec hardware
Module QC
Module QC tools
Baseline for Pixel Module QC
LLS testing
Recipe for working YARR + FELIX configuration
Running at CERN-SR1, ANL, Frascati, Genova, Lecce, Bern
Strips production
QC *
* Done with ITSDAQ in Stage 1, reproducible results with YARR
ITSDAQ hardware
LLS testing
Running and used in Strips system testing in SR1
FELIX hardware
ITk Detector
Detector
FELIX server
LTI
DCS
Monitoring
Ctl & Cfg
CTP
Data handler
SW ROD
ITk SW + YARR Libraries
YARR
TDAQ

Software Development Model

Slides from previous presentation for completeness and easy reference.

Introduction

- How do we maintain the existing functioning codebase, while actively developing software going towards datataking?
- Additional complication: **Exact requirements for ITk operation are not yet clear, and will only become apparent as we go through this process**
- Core functionality during operation will be datataking, but the more complex part will be reacting to different problematic cases in the detector (e.g. stuck chips from SEE, badly tuned chips, unresponsive chips)

→ **Here we aim to outline a possible path and framework for software development, which could be taken by the coordination group**

- Aim to outline our proposed workflow and best practices for code development
  - Goal is to openly outline the organization of development and practices to which every developer converges for constructive cooperation long-term
  - Also serves as entry point for new developers and gives them a detailed guideline how they can help with the development

Introduction

Based on the development model presented here, we would envision the model to work for ITk as follows:

1. Discussion about requirements *[community level]* (ideally including not just general, but also specific requirements, e.g. error handling) (could be via dedicated workshops)
2. Definition of requirements *[ITk SW coordinators]*
3. Planning for implementation *[Software developers]*
4. Development task assignment *[Software (lead) developers]*
5. The actual work *[Software developers]*
6. Report back to community
7. Sign off *[ITk SW coordinators]*

Core framework definition of ITk online SW

- Core framework should include libraries and applications that allow minimal standalone execution of common readout activities
  - Example: data taking for the purpose of debugging, with data being locally dumped
- Core framework should come as a collection of libraries conforming to common types interfaces → each library correspond to one **component** in the software development model
  - Examples of interface types (and their components): ASICs (RD53A, ITkPixV1/V2, Star), Readout hardware (Spec card, FELIX), Processors (Decoder, Histogrammer, Analysis, Error monitor)
- Core framework and its applications should enable full testing of all software components during continuous integration via unit tests and emulators
- Core framework can be used as a library by high level applications to enable large scale functionality
  - Example: interface to configuration database via microservices

Single vs multiple repositories

- Question of single- vs multiple (core) repository/ies:
  - Advantage of multiple repositories:
    - Clear assignment of responsibility
  - Advantages of single repository:
    - Allow for review in a coherent manner & consistency within the repository
    - Simplifies testing
    - Simplifies release building (e.g. avoids having to wait for all repositories to be updated before building tag)

→ **Would strongly prefer keeping everything in one repository for maintainability**

→ following also the experiences of other software developments within ATLAS, e.g. Athena

Library structure & build system

- Single repository does not mean inflexible
- Possible implementation: Repository set up such that each library has its own folder, and in some cases their own dependencies
- Build system set up such that each component can be **compiled independently**, to the extent possible

→ Build handled by **CMake build system**, where one can **mix and match individual components during compile time based on needs**

→ For example: ITkPixV2+Felix, or Star+Spec-based Readout

→ Allows to pull in external dependencies (e.g. NetIO) when necessary

- Should be able to invoke repository/libraries into high level projects via cmake

Basic needs and requirements

- Main deliverable: **Code release which is always running and stable**
- Support for **both pixels and strips**
- Support for **different needs**:
  - Still support bench-testing, module QC
  - System testing during integration
  - Data taking in the detector
- Accomodating for **different kinds of developments**:
  - High-priority short-term fixes
  - Short-term new features and development
  - Long-term developments targeting architectural restructuring (e.g. going towards operation)
- Need to have a well-defined workflow for coherent development on top of the existing codebase

Roles within code development

Different kinds of MRs require different kinds of approval based on their urgency and complexity, handled by different kinds of developers

- **Lead devs:** Experts with detailed knowledge of all components, as well as the big-picture of the software development [2-3]
- **Component managers:** Expert with detailed knowledge in a particular component (e.g. Pixels, Strips, FELIX), available to provide feedback on code on a reasonable timescale [at least 1, ideally 2 per component]
- **Developers:** Main bulk of contributors, submitting merge requests, and encouraged to participate in code review
- **New developers:** very welcome, development model should be aiming to provide an easy entry point through good documentation, MR templates, to do list etc (see also later slide about engagement)

Workflow: Branches and Releases

- Software released as **tags**
- New tag released for each major update to the code
  - Gitlab milestones should be used to plan releases
- Latest tag corresponds to **latest main branch**, and should be a thoroughly tested version of the code
  - This is the version to be used by any production system by default
- **Development branch** is the latest stable* version of the code, which is operational, but may not yet have been thoroughly tested for all use cases (*stable meaning passing CI)
- Any development work should be done based on the devel branch, **with the intention of merging it back into the main branch** (on a reasonable timescale)
  - Try to follow the rule of one MR equating to one feature/change, to allow for efficient review and bookkeeping

Usage of Git issues

- Git issues should serve as the primary task list for developers
- Should supply templates for **types of issues**: support request, bug report, feature request, discussion
  - Templates should auto assign to lead dev or component manager who are requested to either further assign the issue and to adjust labels/milestones/epics
  - Any user should be able to create an issue, specifically for a support request or bug report
  - Opening issues to report bugs is strongly encouraged and could provide opportunities for the reporter to get involved in development if desired
- Add a specific label for tasks that can serve as good **starting point for new developers** (label = starter task)
  - Starter tasks come with an additional more detailed description within the issue
  - Ideally self-contained
- Issues should be assigned to a release when set to “doing”

Issues and merge requests, code review

- Ideally every **merge request should have an associated issue**
  - The merge request should mention the issue # to link them
  - Consider adopting components of this workflow to automate release notes (LSST example)
- The **issue should be used for general discussion** or planning purposes
- The **MR** should (ideally) only be used for **code review**
  - Code review can (and is encouraged) to generally be performed by any developer, but final approval lies with the component responsible/lead devs
  - Human code reviews should maintain a balance between thoroughness to ensure high-quality code and the need for timely releases
  - Code review should be based on common definition of style: e.g. cpp best practices
  - Where possible code review specifically for style should be aided by tools (pre-commit tools should be used, e.g. clang-tidy, clang-format)

Workflow: Branches & Merge requests

- Developments are expected to occur through **merge requests** in the common repository
- Provide **guidelines and templates** for MRs (i.e. as done in Athena), to make clear what information is useful to have to speed up review and make entry barrier lower for newcomers
- Use **automated tools** and testing to lower review burden and speed up MRs
  - Merge requests that do not pass CI are considered “Work in progress” and will not be reviewed/merged
- Management/categorization of MRs/Issues through **labels**:
  - Category of MRs (bug fix, simple, complex, architectural)
  - Components (see earlier slide)
- **Components** specify which exact part of the code is affected:
  - Front-Ends (ITkPixV1, ITkPixV2, Star, ..)
  - Hardware interface (Spec, FELIX, ...)
  - Data processor (Histogrammer, Analysis, Error handler, …)

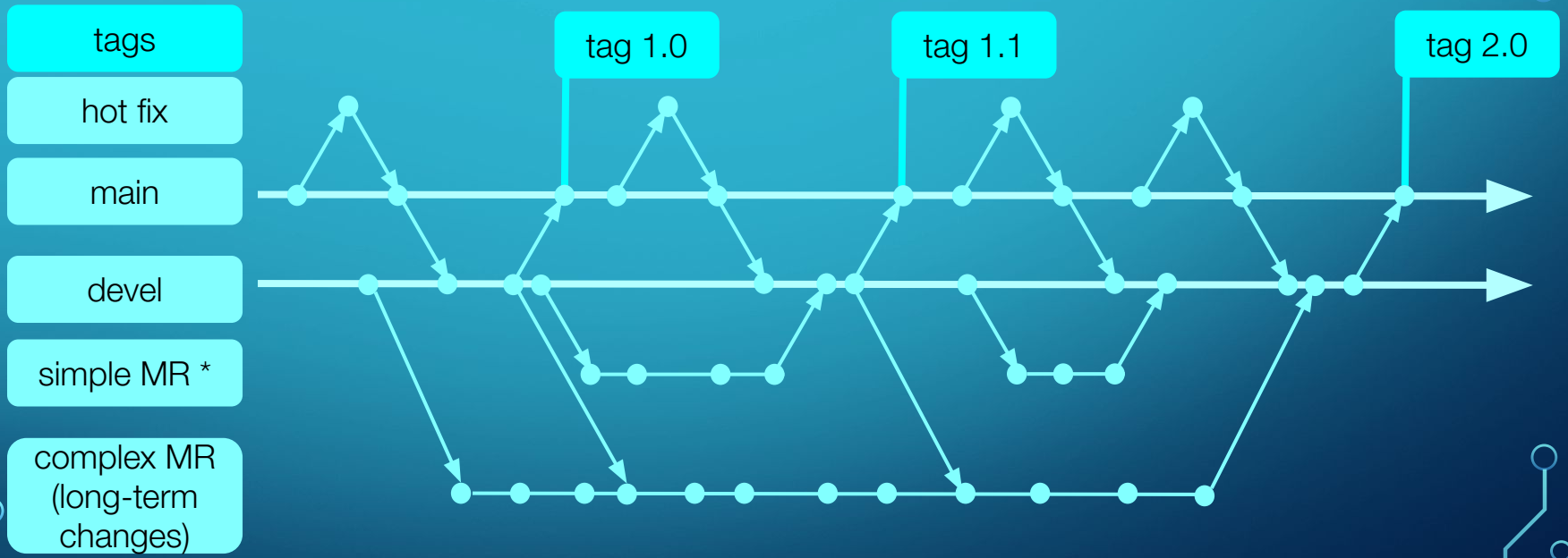
→ **Based on labels and priority, different approaches to code review**

Notes on Git hygiene

- Git should not be used like a “quicksave”
- Users should actively strive towards having **meaningful commits that are cherry-pickable**
  - Squash commits that do not pass CI or are not “standalone”, e.g. avoid “correcting typo” commits
  - This does not mean to squash all commits in a merge requests, it many cases it makes sense to have multiple commits in a merge requests to be able to revert individual changes

  → e.g. using “git commit –amend” or “git rebase -i” to change local commit history before pushing
- See Athena workflow: https://atlassoftwaredocs.web.cern.ch/gittutorial/git-commit/
- Commits should be associated to a specific, not a common account
  - Can be enforced on the repository level via gitlab

Workflow: Branches & Merge requests



* simple MRs with varying levels of complexity (depending on whether it is touching single components, multiple components, interfaces)

Case 1: Hot fixes

Case 1: Clear issue/bug, easy to solve, only affecting one component

- Affects immediate operation and functionality of the code
- Submitted as ~ single-commit MRs directly to the main branch
- Merge requests should be **subject to immediate review** and be merged within ~2 days* by anyone responsible for code development (lead devs or component managers)
- Example: Bug found in ITkPixV2 register read function (e.g. typo)
- In some cases, discussion of the issue will uncover some additional follow-up actions, e.g. where a feature could be implemented in a nicer way, but would take additional time

  → Solution: Merge the quick fix as to fix the bug, and open an issue assigned to the same developer
- Bug-fixes should come with unit test (potentially in separate MR if fast reaction is necessary)
- If applicable, create new minor release after MR is merged & rebase devel branch on main

*time for initial response, if not mergeable with initial response might require more time, to be labelled as such

Case 2: Simple MR

Case 2: Simple bug fixes or new features → covers a broad range of cases, with simple MRs being merged quicker than complex ones:

- Developments generally made based on devel branch, and merged back into devel for thorough testing before merging back into main
- Author is responsible to address any issues that arise during code review

2.1: Single component, no interface:

- MR affecting one component, and no interfaces
- Merged after approval by component manager on a timescale of 1-2 weeks
- Presented (also post MR) in developer meeting if it constitutes a major change

2.2 Multiple component, no interface:

- MR affecting multiple components, and no interfaces
- Requires approval from the component managers of all affected components (e.g. both strips and pixels), merged on a timescale of ~2 weeks
- Presented in developer meeting if it constitutes a major change → up to component manager to decide if presentation is necessary

Case 2: Simple MR

Case 2: Simple bug fixes or new features → broad range of cases, simple MRs merged quicker than complex ones:

2.3 Multiple components & interface:

- MR affecting multiple components, and some interfaces
- Requires **approval from the component managers of all affected components** (e.g. both strips and pixels) & **at least one lead dev,** due to interaction with interfaces, merged on a timescale of ~2-4 weeks

→ In addition to active developers, also requires prompt review/response time from component managers

- **Expected to be presented in developer meeting**

→ Especially if invasive require presentation before performing implementation to avoid already written code to be discarded

- If presentations are required too often (ideally this should occur rarely), consider a workflow that reduces the number of presentation, e.g. use a Request-for-comments (RFC) to solve the issues prior to having the presentation
- If it seems further discussions seems necessary for a MR to converge, the discussion process should be started early (to reduce refactoring too much code)
- Keep discussion and MRs simple → if discussion moves onto a different topic, open a separate MR (and/or issue) to address this topic to keep review process focussed

Case 3: Complex MR (long-term developments)

Case 3: Medium-term developments which significantly affect one or many components

- Broadly covers everything that introduced major changes, e.g. significant change to existing code block/component, new feature, change in structure or workflow for running the code, architectural changes
- Generally requires **discussion of expected developments beforehand** (e.g. in developer meeting), and specific plan for implementation
- Can be longer-term developments in separate branches (with rebasing towards main/devel), with regular updates in developer meetings
- Workflow:
  - Presented **regularly in developer meetings**, with good validation of functionality and subject to discussion in the meeting
  - Documented in Architecture Design Records (and user documentation if necessary), see slide later
  - Agreed with and approved by lead developers, with additional review from component managers in the MR
  - After additional testing in development branch, likely associated with the release of a new tag

Documentation: User documentation

- **User documentation** provides context and instructions for **how to get a working setup**
  - Handled through readthedocs or similar, separate from Architecture Design Records (next slide)
- Provide user documentation that is **use-case oriented:**
  - Setting up a new system
  - Benchtop testing (e.g. SCC)
  - Module QC
  - Interface with FELIX for system testing
- **Documentation for developers:**
  - Decision documentation via Architecture Design Record (next slide)
  - Code documentation via doxygen

- **Implement updating of documentation into the workflow** for every major change, tracked through git issues mentioned in the MRs

Documentation: Architecture design records

- Propose to introduce **Architecture Design Records**
- Document that captures important architectural decision, along with its context and consequences
- **Simple markdown files** inside the repository, if code changes that is captured by an ADR, ADR should change in the same MR
  - Versioning easily tracked and documentation kept in sync with the code for each release
  - minimize work for developers and keep a clear history “for free”
  - ADR changes with the same MR that changes the architecture
- Complimentary to user documentation, **targeted at active developers**
  - Serve as record for why certain decisions were made, as well as documentation for new developers
  - Available on a separate “architecture” page, which is linked from somewhere in the main user documentation
- Start from documentation of existing components and adopting architecture design records as part of the standard workflow
- Explain rationale behind every decision taken, including context and follow-ups

Channels of communication

- **Gitlab:**
  - Automatic notification to specific groups of developers for issues and merge requests
  - As much communication as possible should flow through gitlab as it allows the **discussion to be closely tied to code development** and also enables an “asynchronous” scheme that could be beneficial for large timezone gaps
- **Mattermost:**
  - Channels for users and developers should exist separately
  - Users tend to first contact developers in case of issues via mattermost, but should quickly be **redirect to a support request** in gitlab issues if they cannot immediately helped
    - Support requests can be further evolved into bug reports or feature requests
- **Developer meeting:**
  - Discuss/brainstorm development plan for major changes (either feature or architecture) or releases
  - Discuss status of development of open major merge requests
  - Discuss merge requests that are “stuck” either due to requiring in-person discussion or due to inactivity
  - Plan upcoming releases
- **Workshops:**
  - Semi-regular in-person workshops with **pre-planned purpose**, would ideally follow new requirements definition or major milestones

Engagement

- **Development model is successful if we have community engagement**

→ aim to build a community that is actively developing software towards a common goal

- Progression of new developers can look as follows:
  - Good documentation allows for efficient training of new developers in existing code (user documentation & architecture design records)
  - First contributions through well documented starter-level issues
  - With experience, contributions to MR & code review, as well as longer-term developments
  - Engagement is not just delivering code, also presenting it → e.g. through status updates at ITk/Upgrade week
  - Usage of pair programming to speed up learning curve (Link about pair programming)
- Advantage of proposed development model is that it **allows contributions on all levels** (ranging from newcomers to experts) through the varying levels of MRs and code reviews

Engagement & Workflow

- Development model also **works well with the workflow proposed by the ITk SW task force**
- Specifically, development can happen on a cycle attached to e.g. the proposed mini-workshops:
  - Define which area of requirements should be addressed (roadmap)
  - Define initial requirements
  - Discuss requirements in workshop to understand which are still applicable
  - Decide which components need to change
  - Component managers then translate requirements into tangible actions (issues), which are then implemented until the next workshop (merge requests)
  - At the following workshop, discuss status of software, as well as any requirements that may need adjustment & come up with new action items
- Software development on such a cycle **allows to keep up momentum and ensure coherent developments**
- Additionally, provides a forum for new developers to showcase their work and get feedback
- Note: Role of component manager in this model overlaps heavily with the expected tasks of the responsibles in the ITk SW organigram, so would work nicely within the existing structure

Conclusions

- Discussed the potential framework in which longer-term development of ITk online software can occur
- **Development occurs through MRs into the ITk online SW core framework repository**
- **Issues to track open items** and code development of code done via merge requests
- Implement a **workflow for approving MRs**, with different levels of complexity, depending on the nature of the MR:
  - Quick bug-fixes affecting a single component → merged quickly by any expert available
  - Simple MR → need to be approved by expert developer from each component before merge
  - Long-term architectural changes → planned changes to the code architecture, with planned changes & regular updates presented in developer meetings, before going through approval process on git
- **Documentation** for developers through architecture design records, and separate documentation for users

Conclusions

- SW development for ITk online software **requires a model that allows it to naturally evolve to its final shape**
  - Requirements for detector operation can not be defined a priori, will change and mature over time and as we gather more experience with ITk
  - A software architecture is closely tied to its requirements and therefore needs to be able to evolve with it
- At the core of development model is an **iterative process that over time will adapt to requirements** set by the detector onto the software
- Coherent set of **workflow procedures** ensures streamlined development of high-quality code, which supports both existing systems and long-term developments for detector operation
- **Community involvement is crucial for successful SW development** and new model achieves this by encouraging contributions within the core SW framework through requirements definition, active development, or code review