OpenLMIS Architecture Project

Contents

[Introduction 1](#_Toc436161450)

[Background and Present Day 1](#_Toc436161451)

[Re-Architecture Objectives 3](#_Toc436161452)

[Architecture Vision 5](#_Toc436161453)

[What is Core OpenLMIS? 7](#_Toc436161454)

[What is Not Core OpenLMIS 8](#_Toc436161455)

[A Word on Forks 8](#_Toc436161456)

[Predecessor for Re-Architecture: 2.0 Stabilization 8](#_Toc436161457)

[Architecture Design Approach – Achieving the Vision 9](#_Toc436161458)

[Option 1: Modularity Only – Bare Minimum 11](#_Toc436161459)

[Option 2: Complete Rewrite 11](#_Toc436161460)

[Option 3: “Horizontal Slice” – Start with Core 11](#_Toc436161461)

[Option 4: Recommended Approach - Core Rewrite via Vertical Slices 11](#_Toc436161462)

[3.0 MVP 12](#_Toc436161463)

[Schedule 13](#_Toc436161464)

[What Is Not in Store for 2016 14](#_Toc436161465)

[Resourcing 14](#_Toc436161466)

[Product Roadmap and Related Software Development Initiatives 15](#_Toc436161467)

[Tools and Project Management 15](#_Toc436161468)

[Risks 16](#_Toc436161469)

[Reference: 2015 Community Meeting Targets/Milestones 18](#_Toc436161470)

# Introduction

The purpose of this document is to describe the need, objectives, design approach and schedule of a major re-architecture effort for OpenLMIS.

# Background and Present Day

In the fall of 2012, with strong ongoing guidance provided by the Tanzania and Zambia ministries of health and JSI through the USAID | DELIVER Project, VillageReach and ThoughtWorks began development on the first release of OpenLMIS. Significant contributions from PATH, USAID, Rockefeller Foundation, the Bill & Melinda Gates Foundation, the UN Commission on Life-Saving Commodities, and others shaped OpenLMIS. JSI customized and extended the initial OpenLMIS code base (v0.9) for national deployments in Tanzania and Zambia in late 2013 under the name “eLMIS.” OpenLMIS has also been deployed in Mozambique, Benin, and Cote D’Ivore.

As new installations of OpenLMIS were developed and deployed, challenges surfaced related to the OpenLMIS software internals. A key challenge was the inability to easily extend the code base, which resulted in a “code fork” between the Tanzania and Zambia eLMIS implementations and the Mozambique and Benin OpenLMIS v1.0 implementations.

In an effort to rationalize the fork, the community agreed to being working towards a common master branch hosted in GitHub. ThoughtWorks created the “2.0” branch based on the eLMIS code line, and kept it updated with updates from both eLMIS and a current project for Mozambique.

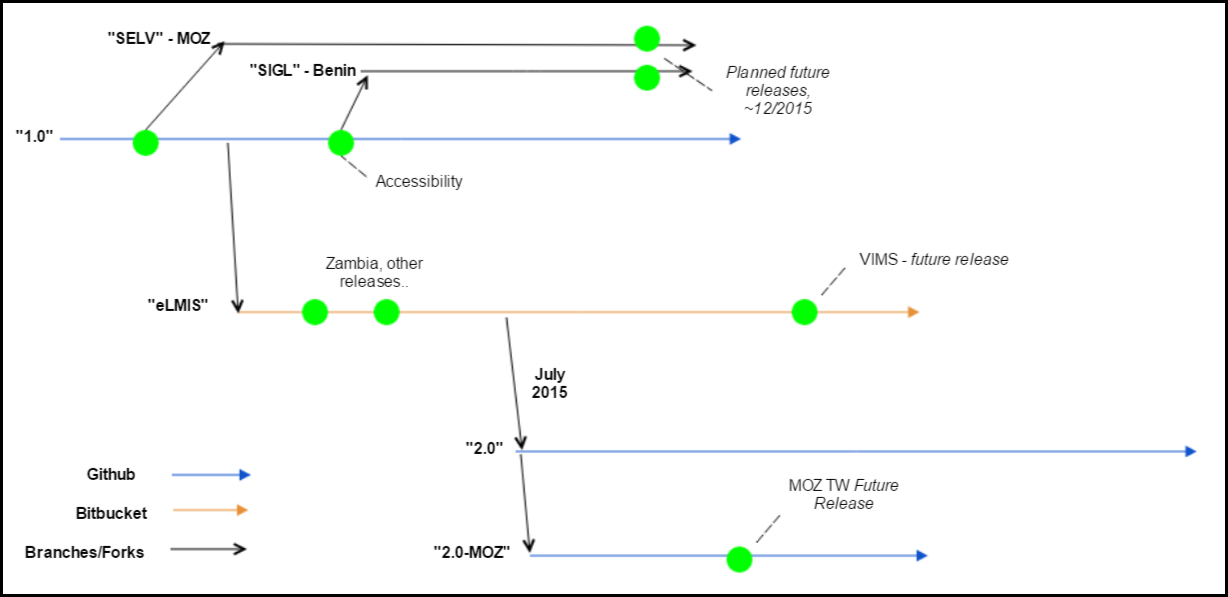


Figure Abbreviated diagram of major OpenLMIS branches

In September 2015, the community met in person for several days, and agreed a single code line is required. Even so, a single master branch does not resolve some of the difficulties that otherwise encourage forking. To summarize just a few of the [pain points](https://openlmis.atlassian.net/wiki/display/OP/OpenLMIS+Pain+Points) identified by the community:

* Monolithic Architecture: the lack of modularity, extension points or similar structures discourages open source development and encourages forking. As such, it is difficult to define a “core” set of OpenLMIS features and services, which may be extended by deployments. Projects often require custom code that should not be included in the “global” OpenLMIS codebase.
* No data collection extensions: Deployments may have specialized data collection needs. Similar to the item above, deployments need a way to collect and report on custom data that can migrate from release to release
* Hardcoded UI: the user interface is not configurable, nor is it easy to brand or change its look and feel. Lack of standard, reusable API to build UI. The UI itself is programmed in an inefficient manner, needlessly consuming precious bandwidth
* Migrations: OpenLMIS does not have a smooth migration/upgrade path
* Extensibility: no structure or practices to extending or modifying workflows, collecting custom data points, etc.
* Developer Productivity: no defined process for contributing to the OpenLMIS code, lack of shared infrastructure and support, outdated component versions (e.g. Java, Angular, etc.)

These pain points must be resolved for OpenLMIS to continue as a viable open source choice for LMIS needs worldwide. Similar open source projects, such as DHIS2 and OpenMRS, have faced similar challenges and at some point in their project trajectory had to undertake similar re-architecture efforts. The changes required are significant, particularly for modularity and extensibility, thus this effort is commonly referred to as the *Re-Architecture* for OpenLMIS. The objectives of the re-architecture effort aims, of course, to address the major pain points.

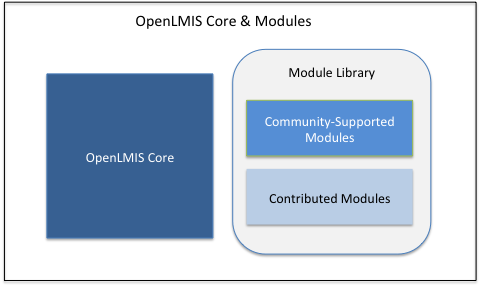
# Re-Architecture Objectives

The goals of the re-architecture effort are to design and implement solutions to a prioritized list of pain points and objectives. The objectives are defined and prioritized by the community and currently [here](https://openlmis.atlassian.net/wiki/display/OP/November+3+2015). The first few are copied below in priority order (the objectives are under very active review, please refer to the link for the updated copy). All work activities will be focused on achieving these objectives in CY 2016.

**Re-architecture Objectives, as prioritized by Product Committee**

|  |  |
| --- | --- |
| 1 | The OpenLMIS user interface can be customized to specific implementations without forking the code base |
| 2 | An extensible framework so that the majority of new code can be built outside of core via extension points. |
| 3 | Legacy OpenLMIS implementations have a migration path to new versions |
| 4 | A tightly controlled OpenLMIS code set (“Core”) that has high test-coverage, high reliability, and changes relatively infrequently. |
| 5 | A straightforward, well-documented way for developers to extend OpenLMIS (create new modules) without forking the code base or modifying the core. This point describes many of the features common to beginner tutorials of a new language or software package, for instance, a "MyFirstExtension" project, and well defined pathways for getting support and making extensions to OpenLMIS. |
| 6 | The community can support the infrastructure needs for “non-supported” code (i.e. creating endpoints) quickly without a lot of bandwidth |
| 7 | OpenLMIS continues to include a user interface, or a reference user interface that can be used as a starting point for customization |
| 8 | Implementers or System Admins can design simple reports without writing code |
| 9 | Extensions to the OpenLMIS code base can be made available to others that may want to use it (just a container, does not imply support) |

In order to support these objectives, we envision creating a logical separation between core code, modules that are supported by the community, and modules that have been shared with the community. This structure allows us to support the key architectural objectives, and follows patterns established by similar open source communities. A diagram of this model, and description of each category is shown below.



|  |  |
| --- | --- |
| Core Code | * Highly Stable and Well-Tested * Changes Infrequently * Regular Release Schedule * Supported by the OpenLMIS Community (bug fixes, etc) |
| Community-Supported Module | * Extension to OpenLMIS Core * Meets Community-Supported Module Standards (testing coverage, coding style, etc. – see wiki) * Aligns with OpenLMIS Product Roadmap * Functionality supports multi-country requirements * Supported by the OpenLMIS Community (bug fixes, etc) |
| Contributed Module | * Any individual or organization can submit and share a Contributed Module * No specific code standards * May be supported by the contributor, not supported by the OpenLMIS community * Often will be created via country-specific implementation projects |

Note that this structure is not intended to imply a particular modular architecture or influence technical design decisions. The purpose is to logically create a difference between these three categories of code.

# Architecture Vision

This section describes the architectural vision for OpenLMIS. We do not expect to achieve this vision within 2016, but it is vital to define the goal to achieve. This vision is intentionally devoid of specific technologies, e.g. Java, OSGi, etc. Such choices will be revealed in the early detailed designs.

Below is the envisioned block architecture and an explanation of each component.

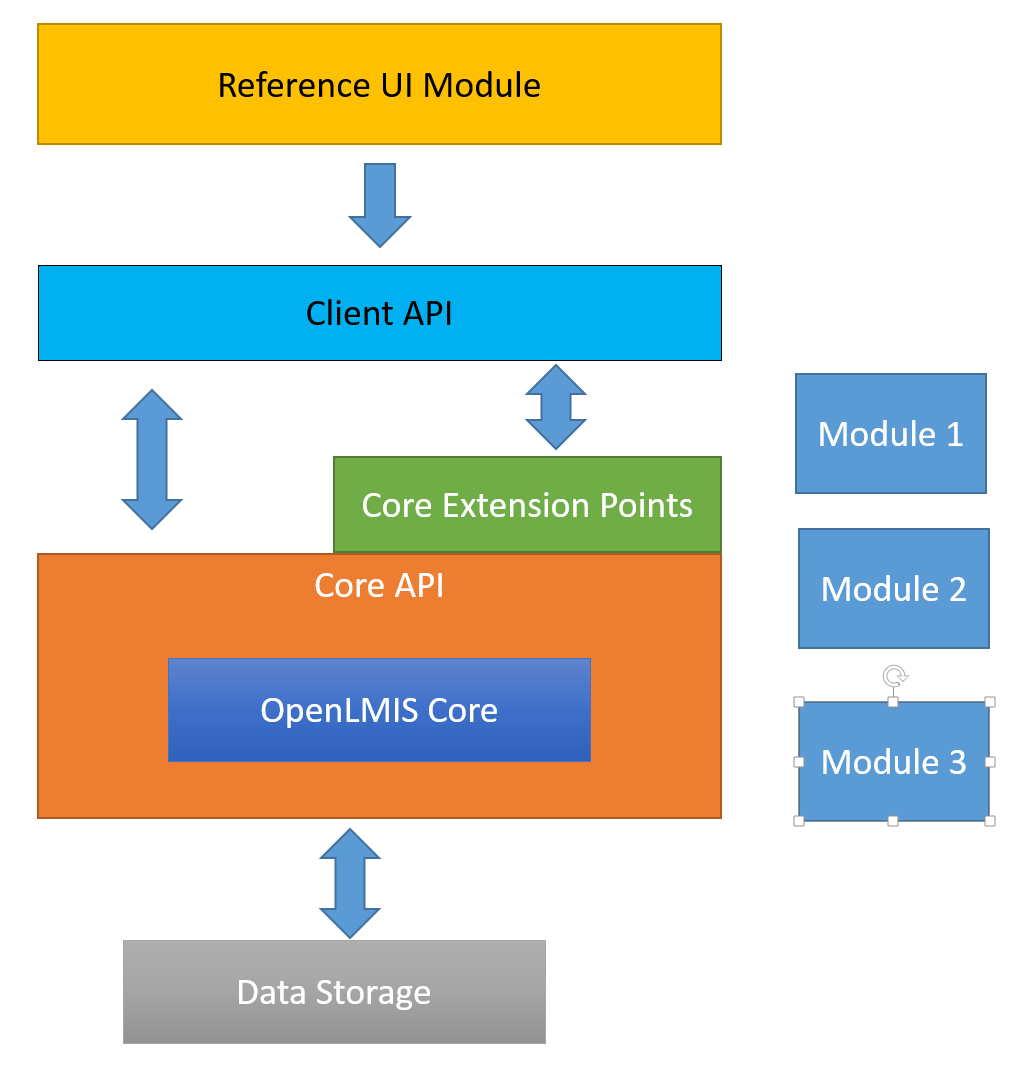


Figure OpenLMIS proposed architecture

**Core:** OpenLMIS will have a core set of services and functionality, which is the essence of the product. It will always be stable, and changes to core are the most scrutinized. Examples of core LMIS functionality includes stock management and requisitions, as well as first-class objects such as Products, Facilities, Users, etc. Core also provides basic application features, such as authentication and authorization, rights and role management, etc. Core features are accessed through standardized APIs and extension points.

Implicit to core OpenLMIS is a ***domain model***, the logical concepts and objects appropriate for OpenLMIS’ problem space. This is discussed in greater detail in the Approach section.

**Reduced Feature Set (as compared to “2.0”)**: a re-architected OpenLMIS, containing *only* core features, will be smaller in size than the current “2.0” branch. The current set of code in the “2.0” branch contains project-specific features that must be removed. This leaves those projects with two options with regards to upgrades:

* Stay with the 2.0 branch, adding enhancements as needed to the project’s own repository, and (optionally) pulling down any changes from the 2.0 master
* If the project wishes to move to the re-architected version, they must re-implement the project changes as one or *modules* (explained below). As we expect the database schema to remain largely the same, a data migration is quite feasible

**Core API**: access to OpenLMIS core is always performed through well-known APIs, insulating the rest of the application from changes to core. This layer of API is distinguished from the Client API – the core API is assumed to be used in-process and requires in-depth knowledge of OpenLMIS internals.

**Client API**: this layer is the public interface of OpenLMIS suitable for all manner of clients, including user interfaces, integrations, point utilities and reporting. It is optimized for low resource needs and is extensively documented for external use.

**Extensions Points**: extension points enable custom behavior within core services. These operate alongside the core APIs. An example of an extension point is the calculation of required stock. OpenLMIS offers restock algorithms based on population or historical consumption. Using an extension point, a module may substitute a third algorithm to meet country requirements.

**Modules**: Modules add functionality not considered part of core OpenLMIS, and are used for adding new features. The feature may be a very project-specific feature, or a broader add-on that is useful to any number of deployments.

Modules are complete packages of functionality, and include its actual logic, data storage extensions, API and UI (the last few are of course optional). It is essentially the layers above OpenLMIS core logic, as depicted below:

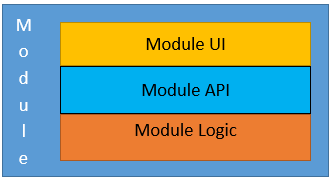


Figure Module Detail

Regardless of its purpose, modules built to OpenLMIS guidelines may be added to any deployment. This fact offers tremendous flexibility and value, enabling various levels of module support. We currently envision three types of module offerings:

* Private. Though technically not an “offering”, it is an option. A private module is one built for a specific project and is not shared with the community
* Contributed. A project may publish its modules for others to use.
* Community. Community modules are those published, maintained and supported by the OpenLMIS community.

It is entirely possible for Contributed modules to become Community supported. A module can, with agreement from the OpenLMIS technical governance, be made part of core.

**Reference UI**: OpenLMIS offers, as a separate module, a reference user interface. The reference UI is functionally complete, meaning it provides access to all core services and provides everything needed for an LMIS implementation. The reference UI interacts with the OpenLMIS server only through the API layer. The API layer is sufficient to build a completely new interface to meet project needs.

Projects that need to customize (versus configure) portions of the reference UI will fork the reference UI module and make the changes. When that project upgrades, it simply refreshes its fork with the latest reference UI release. Please see the forking subsection below for more explanation.

**Data Storage:** Core OpenLMIS defines a standard way to persist and retrieve data, including a means to add additional data storage per project requirements.

Regarding **reports**: reports are essential for any OpenLMIS deployment. Input is required from community on what level of “out of box” reporting OpenLMIS should provide. For now, OpenLMIS will continue offering the embedded Jasper reporting engine. Ideally, any embedded reporting framework tuned for OpenLMIS is implemented as a module. This includes dashboards viewed within the OpenLMIS UI.

OpenLMIS will NOT provide complex analytics or business intelligence capabilities

## What is Core OpenLMIS?

The “core” of OpenLMIS is the set of services and features that compose its essential value, an open-source LMIS solution for low-resource environments. Per the September 2015 Community Meeting, the community agrees with the concept of core and embarked on a group exercise to list out what is and is not core OpenLMIS. The raw results are here: <https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=30638249>. The community must refine and ratify this list. For the purposes of getting started, core items are listed below. We expect this list to be refined by Product and Tech committees.

* Requisitions
* Stock Management
* Program-based (Multiple “verticals” of Programs with their own products, requisition workflows, costing data, etc.)
* Offline Capability
* Essential objects: Programs, Products, Facilities, Users, Supply Chains, Delivery Zones, Requisitions
* Reference Data – out-of-box data essential to OpenLMIS operation. Examples: Stock Adjustment Reasons
* Basic cold chain equipment support
* Data source for analytics (ideally through an interface vs. direct data store access)
* Basic Reporting Module, with out-of-box reports reflecting supply chain good practices

## What is Not Core OpenLMIS

Below are some examples of non-core features. These are best suited as modules.

* Dispensing
* Electronic Medical Records
* Bar coding
* Analytics tool
* Budgeting
* Master Product List
* Disaster Relief & Response
* Features added for specific country deployments, such as:
  + ILS Gateway
  + Vaccine-related workflows (VIMS)

## A Word on Forks

As the community has extensively discussed and agreed upon, we want to avoid forks of OpenLMIS itself. This is what the concept of core, modularity and extensibility is meant to alleviate. One of the original first objectives was to allow for “customization of the UI without forking”. Is this a contradiction?

Core OpenLMIS, that essential set of logistics services and functionality, should not be forked. OpenLMIS’ current monolithic nature encourages forking as it is the only real way to make any kind of customization. By separating out the core, other aspects of the solution can be forked to accommodate project changes, and they rely upon a stable core.

The UI is a good example. The UI should be divorced from OpenLMIS core. Resting on top of common APIs, it becomes a separate module, or even a separate project. When major UI changes are called for, we propose the strategy of forking a proper UI module, and keeping it refreshed from release-to-release.

# Predecessor for Re-Architecture: 2.0 Stabilization

At the September 2015 Community Meeting in DC, it was agreed the branch labeled, “2.0” is not suitable for any new implementation to start from. This is due in part to incomplete features, functionality specific to a country deployment, and missing critical documentation, among others. We anticipate new OpenLMIS projects will start before any significant re-architecture work is completed, so it is critical to provide a stable release from the 2.0 branch.

The community drafted a list of tasks required to bring the branch to release tag status and establish it as the new master branch. This is the first priority for both the community and the re-architecture effort, with an expected finish date of January 15, 2016.

A few of the major items include:

* A shared development infrastructure
* Disabling, or “toggling off” the UI elements of features not deemed core to OpenLMIS
* Updating the database migration scheme, managed by flyway. The chief benefit is easing data migrations, a major pain point
* Published coding standards
* Published Developer Contribution Guide, which includes procedures for adding project-specific sub-modules and how to contribute back to the master branch

# Architecture Design Approach – Achieving the Vision

*At the September 2015 Community Meeting, the technical team discussed micro services as a possible approach for implementing a modular architecture in OpenLMIS. ThoughtWorks proposed a three-month spike to re-implement a vertical slice of OpenLMIS using a micro services approach. Advantages of this approach were a quick "jump start" on the re-architecture and the ability to better understand what it would take (time and budget) to fully re-implement OpenLMIS using micro services. Negatives included the likelihood of needing to fully re-implement OpenLMIS (versus and incremental approach), lack of analysis of multiple alternatives, particularly with regard to options besides micro services, and cost to fund this effort. The vertical slice approach is quite practical and is part of the recommended approach, with additional concept of modeling and building a portion of core.*

There are several approaches to achieve the re-architecture goal. Whichever route is taken, we refer to “**3.0**” as the next releasable version of OpenLMIS. This major number increase emphasizes the new design patterns and reduced feature scope.

We do not expect to achieve the full re-architecture of OpenLMIS and meeting all of the objectives in 2016. As with any software project, there are trade-offs in terms of time, cost, and delivered functionality. For this project, there is a defined time period (current funding runs through May 2017) and budget. In addition, the longer the re-architecture takes, more functionality will be built on the OpenLMIS 2.0 code base that would need to be implemented on the re-architected OpenLMIS, and more new countries may implement OpenLMIS 2.0. Considering these trade-offs, we have set a target completion date of November 2016 to release OpenLMIS 3.0. This timeline needs to be vetted by the community. Current analysis of design alternatives takes this time window as a given, and alternatives have been assessed using this constraint.

For any alternative, the following is **required**:

1. **Reduced Scope**: the 2.0 branch contains features that are not considered core OpenLMIS (See section, What is Not Core OpenLMIS). Per the architecture vision, these features should not be in the base code. Maintaining these features as part of, or alongside, the architecture work would increase the overall effort and timeline. Therefore, **non-core features from 2.0 will be removed or toggled off**. These features will **not** be in the re-architecture scope, and must be re-refactored as a module if existing deployments wish to carry on to 3.0.

This seems to run counter to one of the higher priority Re-Architecture Objectives, “Legacy systems can migrate to new versions”. As stated earlier, data migrations will be easier than feature migrations. Legacy systems that make use of non-core features will likely need to update those project-specific features as modules.

1. **Domain Modeling**: OpenLMIS operates in a specific domain – LMIS systems in low-resource environments – and the core must be modeled to fit. A domain model consists of the logical design of classes, services and extension points. For example, a domain model for Requisitions includes object designs for the major actors and how they relate, the services and APIs available to interact with the model (requisition creation, approval, etc.) and extension points for clients to modify that behavior to fit specific project needs.

Domain modeling is a critical first step, as it informs APIs, extension points, etc., which are required to meet the objectives of modularity and extensibility. The domain model incorporates the knowledge gained since the first versions of OpenLMIS, and can better adapt to future needs.

Domain modeling will also help rationalize emerging concepts in OpenLMIS. Stock Management is a prime example. Added recently, it doesn’t integrate with congruent functionality in say Requisitions, which requests stock in/out information from the user. Stock Management could inform this as part of core, therefore building Stock management into core should be part of the domain model.

1. **Horizontal Expansion, i.e. “modules”**: OpenLMIS requires a formal horizontal expansion mechanism, a way to add additional functionality that is housed in a separate code repository and does not modify core code. Modules should contain the logic, data storage extensions/migrations, and UI additions. Some pieces will require design and code support, such as data storage extensions.

How modules are “added” is largely a technology choice, and one available now to OpenLMIS is in the form of gradle sub-projects. This approach adds modules at **build time**. To support this, the new Contribution Guide will formalize the gradle sub-project approach as the only community-approved means for adding project-specific functionality.

1. **Data Storage**: Fortunately, the database schema is stable and appropriate for OpenLMIS and thus little change is required. Keep the schema intact opens the possibility of data migrations from previous OpenLMIS deployments. The architecture must address extending data storage to projects. Most projects have new data points to store, and the architecture design must accommodate this need.

The approach options that have been considered include:

1. Modularity Only – bare minimum to achieve a level of modularity
2. Complete rewrite – start with a blank slate, preserving only the schema/data model
3. Layered Approach (“horizontal slice”?)– start building core, and insulate changes to upper layers with a proxy layer
4. Vertical Slices – inline rewrite starting from existing code base

## Option 1: Modularity Only – Bare Minimum

This approach starts with the concept of modularity. Using the existing gradle sub-project functionality, or something more advanced, we formalize the concept of adding functionality at build time in a project-specific repository.

While quite possible, even with today’s OpenLMIS, it does nothing to resolve the extensibility problem. A separate module, in order to add value, may need to modify a core concept in order to function. Without extension points, this is only possible by modifying core code, a change that affects all future deployments.

To take a real-world example, OpenLMIS requisition approval workflow did not meet VIMS’ requirements. The approval workflow offers little extensibility, so the project faced the choice of either modifying the workflow – affecting all future OpenLMIS projects – or building an alternative. This violates the objective of a tightly controlled core, and as such, we dismiss this option as a final solution.

## Option 2: Complete Rewrite

The blank slate approach frees us from working with the confines of existing code and structures. With this freedom comes speed and the ability to apply all lessons learned from previous OpenLMIS work without trying to shim new concepts into an existing code base.

The downside is that no value can be derived until the core functionality is 100% complete. It is unlikely this can be achieved in 2016 with current resources.

## Option 3: “Horizontal Slice” – Start with Core

In this approach, we update core OpenLMIS functionality at the lowest level possible – its domain model, implementation and persistence. The layers above it – any additional business logic and UI – remain the same. Between the new core and the legacy layers is a temporary proxy, which translates the old layer to the new core. After the core is complete, the next layer is added (APIs), the old proxy is removed, and a new one translates existing logic and UI to the new API layer. Proceed in this fashion until the UI itself is refactored as the reference UI.

## Option 4: Recommended Approach - Core Rewrite via Vertical Slices

This approach works with the existing code base. The core is built up by replacing functionality with “vertical slices”. For example, the requisition functionality is replaced with a slice resembling the architecture vision, including a domain model, core logic, APIs and interface points, and updated UI. This is the recommend approach, and reflects the community valuing existing projects and carrying those forward (the rewrite approach values future projects, building a system that is extensible from the start).

This is an “inside out” approach, moving from the most stable layers to those that are more dynamic. It focuses on what is most important first – core LMIS functionality – and layers on APIs, extensions, management and UI.

The risk in this approach is it may take considerably longer, as existing code that interacts with the domain will need to be refactored. Updating or rewriting the UI layer to interact with new core pieces could be especially difficult. There is a chance that the vertical approach incurs too much overhead, and reverting to the “full rewrite” option becomes the most effective choice. As this is a critical decision point, we will time-box the vertical slice approach, with an evaluation and decision point due in March 2016.

The general approach to designing and implementing architecture changes is as follows:

1. Assemble list of OpenLMIS pain points (*Done!*)
2. *Assemble a list of what is an in note OpenLMIS core (Done!)*
3. Establish a prioritized list of objectives for the re-architecture effort (*underway)*
4. Continue and complete the 2.0 stabilization effort
5. Community agrees on non-core features that will not be maintained as part of 3.0 work
6. Choose a core concept to begin the vertical slice
   1. Design and create its domain model. The end result is a functional specification, a working set of classes, and a portion of core API to interact with the domain.
   2. Refactor the new core model into existing logic
   3. Design and write a portion of Client API suitable for UI and external clients
   4. This activity is time-boxed to 3 months
7. Evaluate, make adjustments, and start with the next slice of core functionality
8. In parallel, start design and implementation iterations to address other aspects of the architecture, such as extensible data storage, modularity, and UI branding
9. Continue building slices
10. Stabilize and release 3.0

During the first slice iteration, technology choices are also made. As an example, to use a REST-style API for the client API. As we are operating within existing OpenLMIS code, we will continue to write core code in Java, use Postgres as the data store, etc.

Our goal is to meet the modularity/extensibility objectives by November 2016 as version **3.0.** This will contain one or more high-priority vertical slices and modularity support. While the full architecture vision will likely not be complete, an incremental releases provides several benefits:

* Provides a much better starting point for new projects, with cleaner upgrade paths to future versions
* New extension points are available for use
* Prevents a stale code base. Besides re-architecture, bug fixes and minor improvements will likely be made

## 3.0 MVP

* Module support – technical. Includes all support to build in standalone logic, UI and data storage extensions. Includes documentation and a sample module.
* Core slices. These include a domain model, logic, Core API, extension points and client API. In rough priority order, the core slices to include are below.
  + Requisitions
  + Stock management
  + Programs
  + Essential objects: Programs, Products, Facilities, Users, Supply Chains, Delivery Zones, Requisitions
  + Reference Data – out-of-box data essential to OpenLMIS operation. Examples: Stock Adjustment Reasons
* *What about data storage extensibility?*
* Client API documentation
* (stretch) Separate UI module

# Schedule

This section describes a high level milestones for the re-architecture effort. A detailed schedule will be published and updated on the wiki.

At the 2015 Community Meeting, the Technical community drafted a list of milestones for 2016, which this schedule takes into account. That original list is found at (<https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=30638284>). For simplicity, the schedule below shows only major milestones.

This schedule assumes full resourcing from the OpenLMIS global software development team without contributions from other software development partners. If other teams have software development resources to contribute pro bono, the schedule will be modified.

We aim to achieve the modularity and extensibility requirements by November 2016. We do not anticipate to have achieved the all the re-architecture objectives. We recommend at least one more architecture-focused released in Q1 2017. Before 3.0 is released, the Product and Tech community will map out the scope of the 3.0 follow-on release, and whether it contains new OpenLMIS core features, community supported modules, continued architecture/infrastructure work, or a blend of the above.

| **Milestone** | **Benefits** | **Date** |
| --- | --- | --- |
| Shared Development Env. | Common build, CI, tools, etc. | December 2015 |
| Architecture Design approach approved by community | Major designs/decisions reflect collective expertise | December 2015 |
| 2.0 stabilization | Starting point for new projects  Improved data migration | January 2016 |
| Coding Standards/Contribution Guide v2, standards enforced | Community-enforced standards for updating master branch | February 2016 |
| Shared 2.0 Demo server | Public server for demos and reference | February 2016 |
| Domain Model v1 | Functional Spec.  Classes for portion of model.  Beginning of Core API. | March 2016 |
| Client API v1 | Accesses domain model v1. | March 2016 |
| Continue Slicing or Rewrite, Evaluate and Decide; Determine 3.0 scope |  | March 2016 |
| Monthly updates and demos | Display progress, highlight critical decisions and changes | March through May 2017 |
| Freeze 3.0 scope | Finalize 3.0 scope for release prep/planning. | July 1, 2016 |
| Determine 4.0 scope | Prioritized list of features for next release; enables Product team to gather requirements while 3.0 winds down | August 2016 |
| 3.0 Release | Higher-priority core and APIs complete.  Full module support.  UI module ready (*stretch).*  *Identify scope for next release.* | November 2016 |
| 4.0 Release (*recommended*) | Continued architecture/infrastructure updates, based on remaining work and feedback from 3.0. | March 2017 |

*Need to fit a UI analysis and design strategy into the schedule: is this built along with core slices on the new Client API? What sorts of common services are needed (e.g. session management), and how to encourage low coupling (e.g. CSS, common API calls, common form structure)?*

## What Is Not in Store for 2016

Give the large scope and current resources, we do not anticipate to achieve the full re-architecture vision. Some items not scheduled for 2016 include:

* All core features may not be “sliced” into core
* We will not achieve a complete reference UI or “easy” UI customization.
* No new embedded report design capabilities. OpenLMIS already has Jasper built into it for projects to use for reports and dashboards. Proposal is for reporting capability to be another module.
* No new feature support

# Resourcing

The OpenLMIS global software development team (housed at VillageReach), funded by the Bill & Melinda Gates Foundation OpenLMIS grant, will be fully allocated to this project over the duration of 2016. This includes three software developers, .5 product manager/BA, and .8 software development manager.

The global team welcomes in-kind software development contributions on the re-architecture effort from other community partners.

Participation by the Governance, Product and Technical committees is crucial for success. The Technical committee is responsible for the detailed design and implementation. Major questions involving features, functionality or administration of OpenLMIS (such as which features to “toggle off” for global OpenLMIS, or support for existing deployments) should involve the Product committee.

# Product Roadmap and Related Software Development Initiatives

At the September 2015 community meeting, it was agreed that OpenLMIS re-architecture is the highest priority product roadmap item for OpenLMIS. As such, all global software development team resources will be allocated to this project in 2016. This naturally means that other roadmap items will not be built out during 2016. Thus, in general, we do not expect expansion in the OpenLMIS feature footprint to be generated by the OpenLMIS global team in 2016.

That said, individual implementations projects or other funded initiatives may generate software development that will not be part of OpenLMIS Core. Projects that are already planned for 2016 include:

* VIMS (JSI, CHAI, PATH) – continued development of Tanzania vaccine management functionality.
* Malawi OpenLMIS implementation (JSI, VillageReach) – Implementation of OpenLMIS 2.0. Development of offline R&R feature, and possibly reports, on top of 2.0 code base. These new features will not be included in OpenLMIS 3.0.

These new projects should begin with the latest release tag (2.0), and any new functionality should be created as modules as extensively as possible. As will be detailed in the planned Contribution Guide, projects should be maintained in a separate branch or repository, and pull down changes from the OpenLMIS master in github. Projects should not assume their updates, especially changes made to forthcoming core logic, will be accepted back into master.

Because of the VillageReach global team’s full allocation on the re-architecture, the team will not be available to conduct software development for specific implementation projects until the re-architecture is completed. VillageReach may leverage outsourcing to established software development partners or OpenLMIS community members to complete any required software development on OpenLMIS 2.0 while the re-architecture is in progress.

# Tools and Project Management

To facilitate development and collaboration in a distributed community setting, OpenLMIS partners will use common tools and procedures:

* JIRA: defect/bug tracking; definition and assignment of work items.
* Confluence: Wiki; collaboration, publication
* Google Groups: distribution lists
* Jenkins: CI server
* Sonar: Static Code quality analysis
* GitHub: source code control

All work items are defined and tracked in JIRA, and contributors are expected to use JIRA to assign tasks, update task status, log bugs, etc.

The high-level schedule is posted and versioned on Confluence.

The Technical Committee is responsible for the development effort and project management. Technical will provide periodic progress updates to the Product and Governance groups.

# Risks

| **Risk** | **Likelihood** | **Impact** | **Mitigation** |
| --- | --- | --- | --- |
| Lack of Community Involvement |  | Decision-making does not representative; reduced staffing for work | * VillageReach has several full-time personnel dedicated to this effort for 2016 * A prioritized list of objectives helps us deliver most important items earlier * Dev continues to add support, helps new contributors come up to speed |
| Lack of consensus (forks continue) | Low | Effort spent on “one-of” deployments that are difficult-to-impossible to update to newer OpenLMIS versions | * Constant collaboration amongst the OpenLMIS community |
| “Analysis Paralysis” | Medium | Too slow of a pace to meet milestones | * Publically define work items, owners, due dates, dependencies * Hire architect for short term assistance |
| Feature-level migrations not possible. Maintaining country-specific features in the midst of re-architecture may prove unfeasible | Medium | Existing deployments wishing to migrate to 3.0 must refactor custom functionality as modules |  |
| Refactor overhead (adding new concepts in old code) shows vertical slice approach is not feasible | Medium | Prolongs schedule, adds complexity | * Decision point in early 2016 on the effectiveness of the slice approach |
| Competing Priorities/Project vs. Product (project work distracts from re-architecture) | High | Lost momentum; 3.0 release delayed | * Focus on highest priority features |

# Reference: 2015 Community Meeting Targets/Milestones

(*Following is a copy of the technical milestones hoped for 2016 as identified at the September 2015 Community Meeting. Original source*: <https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=30638284>)

**Q4 2015:**

* Master/upstream branch established
* Shared infrastructure for devs (CI, sonar, etc.).  Ideally the Sonar results are published
* Code standards v1 (first steps to get it going)
* Start Architecture spike

**Q1 2016:**

* Code standards evolved (ongoing discussions anticipated)
* New standards enforced (via automated checks)
* Contribution Guide v1
* Architecture spike completed, reviewed, and decisions made.
* Shared demo environment (implies demo DB)
* Public community support forum (simply a public DL.  No service level committed to)

**Q2 2016:**

* Published technical documentation begins

**Q3 2016:**

* Extensible architecture ready - new implementations can write extensions, and have a path to contribute back to core