OpenLMIS Architecture Project

Contents

[Introduction 2](#_Toc442869436)

[Background and Present Day 2](#_Toc442869437)

[Changes since v1. 3](#_Toc442869438)

[Re-Architecture Objectives 3](#_Toc442869439)

[What is Core OpenLMIS? 3](#_Toc442869440)

[Architecture Vision 4](#_Toc442869441)

[First Step for Re-Architecture: 2.0 Stabilization 6](#_Toc442869442)

[Architecture Design Approach – Achieving the Vision 6](#_Toc442869443)

[Re-Architecture Options 7](#_Toc442869444)

[Incremental Approach 7](#_Toc442869445)

[Schedule (Incremental) 9](#_Toc442869446)

[Complete Rewrite 9](#_Toc442869447)

[Not Included in 2016 Scope 10](#_Toc442869448)

[Resourcing 10](#_Toc442869449)

[Product Roadmap and Related Software Development Initiatives 10](#_Toc442869450)

[Tools and Project Management 11](#_Toc442869451)

[Risks 11](#_Toc442869452)

[Reference: 2015 Community Meeting Targets/Milestones 12](#_Toc442869453)

[Appendix A – Core and Modules 14](#_Toc442869454)

[Appendix B – Re-architecture Options (not-prioritized) 14](#_Toc442869455)

[Option 1: Modularity Only 14](#_Toc442869456)

[Option 3: “Horizontal Slice” – Start with Core 14](#_Toc442869457)

[Option 4: Core Rewrite via Vertical Slices 15](#_Toc442869458)

[3.0 MVP 16](#_Toc442869459)

# 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, an open-source logistics management information system (LMIS). 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,” and OpenLMIS has also been deployed in Mozambique, Benin, and Cote D’Ivoire.

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 address the fork, the community agreed to begin working toward a common master branch hosted in GitHub. ThoughtWorks created the “2.0” branch based on the eLMIS code line, and kept it current with updates from both eLMIS and an existing project for Mozambique.

During the all-community meeting held in September 2015, three governance groups were formed to help guide and manage the activities of the OpenLMIS community. The Governance, Product, and Technical groups meet regularly to discuss and find consensus on key issues relating to OpenLMIS development and management.

In September, the community also agreed that a single, “core” code line was required. This alone does not resolve many of the difficulties that may cause forking, and three of the primary pain points identified by the community are below.

* 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. Projects often require custom code that should not be included in the shared OpenLMIS codebase
* No data collection extensions: deployments may have specialized data collection needs, and need a way to collect and report on custom data that can migrate through releases
* Hardcoded User Interface (UI): the current UI is not configurable, nor is it easy to brand or manipulate. OpenLMIS also lacks a standard, reusable Application Program Interface (API) to build UI

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 comparable 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.

## Changes since v1.

Highlighted incremental and rewrite approaches, moved others to appendix; added Reference Application in Architecture Vision**;** Updated Schedule section; general cleanup.

**Pending items.** References to 3.0 may no longer be accurate under a continuous model; Better language around feature de-scoping; show how team could encourage others to write or update feature to Do the Right Thing

# Re-Architecture Objectives

The overall objectives are to better **enable contribution** and **maximize shared benefit** by promoting code reuse and transferrable customizations, while maintaining a stable OpenLMIS core. This includes maintaining processes, policies and technical support for the contribution of features and fixes to OpenLMIS. Transferrable customizations, also called “modules,” are a mechanism for projects to build non-global features that may be applied to future versions of OpenLMIS, thereby easing the upgrade or migration process.

The Product committee has established a list of re-architecture objectives [here](https://openlmis.atlassian.net/wiki/display/OP/November+3+2015). Key objectives are listed below in order of priority (and are currently under active review), and proposed solutions to these objectives are explored later in the document.

* The OpenLMIS user interface can be customized to specific implementations without forking the code base
* Develop an extensible framework so that the majority of new code can be built outside of the core via extension points
* Legacy OpenLMIS implementations have a migration path to new versions
* Instate a straightforward, well-documented way for developers to extend OpenLMIS and create new without forking the code base or modifying the core

## 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. During the September meeting, the community agreed with the concept of core and embarked on a group exercise to list the primary aspects of core OpenLMIS. The raw results can be found [here](https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=30638249). While the community must refine and ratify this list, initial points of agreement on key aspects of core are listed below:

* 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
* 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

# Architecture Vision

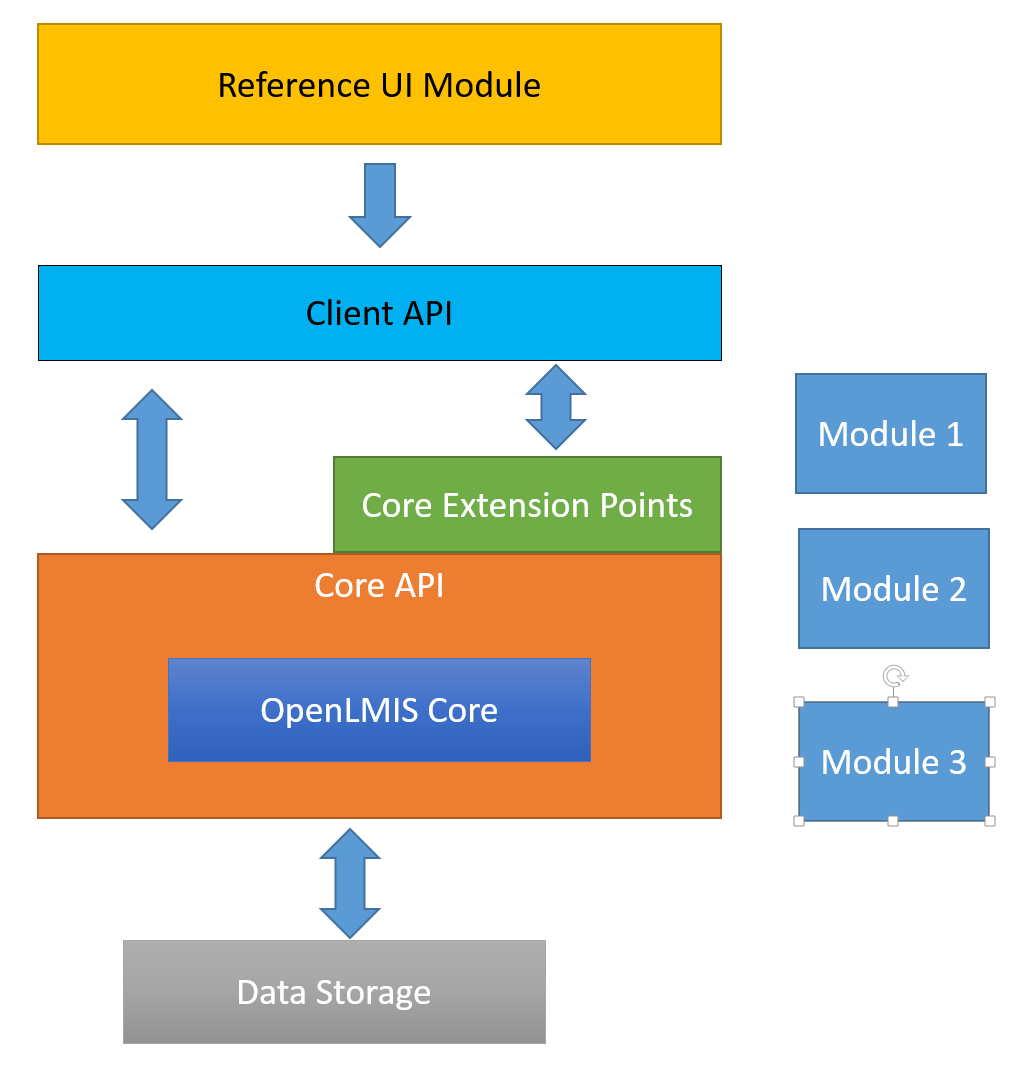


Figure OpenLMIS proposed architecture

Moving forward, the vision for the re-architecture of OpenLMIS includes a number of key developments critical to the functionality and flexibility of the product. These terms may be considered the primary components of the OpenLMIS vision, and figure greatly in the overall structure of the product as it evolves. Figure 1 presents the architecture vision.

**Core.** OpenLMIS will have a core set of services and functionality, which is the essence of the product, and which will include common LMIS components such as stock management and requisitions, as well as first-class objects such as Products, Facilities, and Users. Core also provides basic application features, such as authentication and authorization, rights and role management. Core features are accessed through standardized APIs and extension points.

**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.

**Extension 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 actual logic, data storage extensions, API, and UI (the last few are optional). Modules act as the layers above OpenLMIS core logic, as depicted in figure 2.

Some examples of features best suited for a module are:

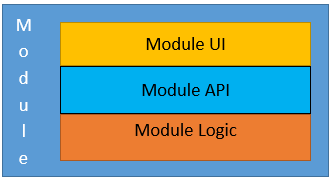


Figure 2 Module Detail

* Dispensing
* Electronic Medical Records
* Bar coding
* Analytics tool
* Budgeting
* Master Product List

Appendix A lists the further characteristics of core and modules. A module can, with agreement from the OpenLMIS technical governance, be made part of core. It is entirely possible for contributed modules to become community supported.

**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 as well as 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.

**Reference Application.** To complete the separation started with the core and the reference UI, OpenLMIS delivers a reference application. Treated as a separate project, the reference application is composed of core, the reference UI, and any community-supported modules deemed appropriate. The reference application contains all the logic for a modern web application and will end up looking mainly like OpenLMIS today. The advantages of a separate reference application are a different level of modularity and protection of core from changes more likely to occur (for example, a change to the application’s authentication scheme).

**Data Storage and Reports.** Core OpenLMIS defines a standard way to persist and retrieve data, including a means to add additional data storage per project requirements. Reports Regarding are essential for any OpenLMIS deployment. Input is required from the community on what level of “out of the box” reporting OpenLMIS should provide. For now, OpenLMIS will continue offering the embedded Jasper reporting engine. Ideally, any embedded reporting framework tuned for OpenLMIS will be implemented as a module. This includes dashboards viewed within the OpenLMIS UI. OpenLMIS will NOT provide complex analytics or business intelligence capabilities.

**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 as, by definition, it does not belong in core. If the community desires these features to be included, either as core or as a module, it would be a good collaborative effort.

# First Step for Re-Architecture: 2.0 Stabilization

While OpenLMIS is currently operating under the 2.0 branch, the September community meeting determined that the branch is not suitable as a base for any new implementations of OpenLMIS in its present form. This is due in part to incomplete features, functionality specific to a country deployment, and missing critical documentation. Any new implementations of OpenLMIS will likely start before significant objectives of the re-architecture work are achieved, therefore stabilizing the 2.0 branch is a top priority.

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 completion date of February 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
* 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, including coding standards, quality expectations, etc.

The contents of the branch labeled 2.0 will be merged into “dev”. This is later merged into “master” and granted a release tag of 2.0.

# Architecture Design Approach – Achieving the Vision

There are several approaches to achieve the re-architecture goal. Achieving the full re-architecture of OpenLMIS and meeting all of the objectives in 2016 is not likely to occur. For this project, current funding runs through May 2017, with a defined budget. By taking an incremental approach for continuous development delivery of updates, the community will be able to coordinate and capitalize on knowledge gained as the re-architecture progresses.

For any approach, the following is required:

1. **Reduced Scope.** The 2.0 branch contains features that are not considered core OpenLMIS (see pages 4 and 5 for features better suited for modules). Per the architecture vision, these features should not be in the base code, and 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 the functionality is desired.
2. **Domain Modeling**. OpenLMIS operates in a specific domain – LMIS systems in low-resource environments – and the core must be modeled to fit this mandate. 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.

1. **Horizontal Expansion, i.e. modules.** OpenLMIS requires a formal horizontal expansion mechanism as 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 needed. 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**. Little change is required for the existing database schema, which is stable and appropriate for OpenLMIS. Keeping the schema intact opens the possibility of data migrations from previous OpenLMIS deployments, although the re-architecture must address extending data storage to projects. Most projects have new data points to store, and the architecture design must accommodate this need.

In September 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 are the possibility of a quick "jump start" on the re-architecture and the ability to better understand level of effort (LOE) and budget requirements to fully re-implement OpenLMIS using micro services.

This vertical slice approach is quite practical, with the additional concepts of modeling and building a portion of core. Challenges include the lack of analysis of multiple alternatives, particularly with regard to options besides micro services, and cost to fund this effort. Discussions on this point are ongoing.

# Re-Architecture Options

A number of options were considered as the recommended path to achieving the re-architecture vision for OpenLMIS. Five total options were explored, and a full explanation of those not prioritized is included in Appendix B. The two leading options are explored below.

## Incremental Approach

This approach provides the most agility within the community. New projects may begin to leverage new architectural features as soon as they are available. Architectural features will be implemented in priority order, and release tags will be added as major features are completed.

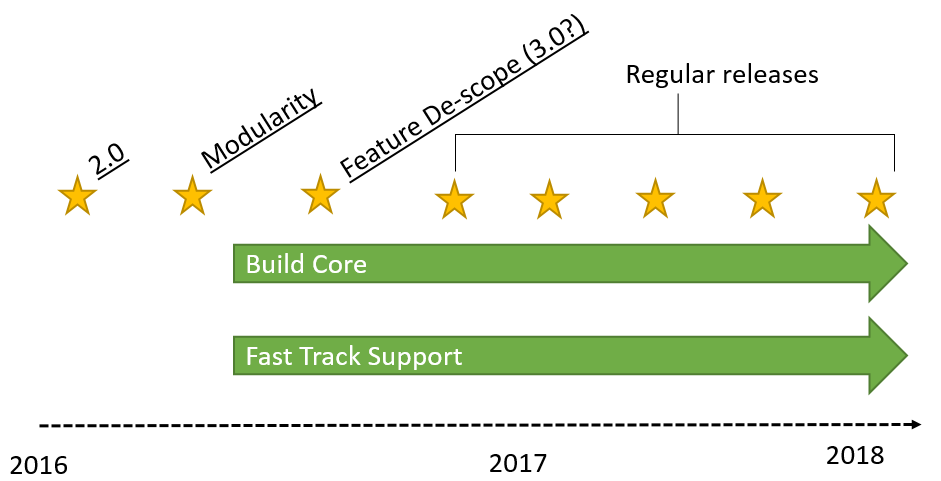
Priority list for an achievable approach:

1. **2.0 release.** This release provides a solid foundation for new implementations. It implements an improved mechanism for the community to contribute to OpenLMIS, mainly through agreed-upon standards, procedures and shared infrastructure
2. **Modularity v1.** Provide a first release of improved modularity to address major pain points currently being experienced
3. **Establish core and Remove non-core features (3.0).** The establishment of core, domain modeling, and extension points is an ongoing process spread amongst successive releases. High-priority core items include requisitions, stock management, Programs, Essential Objects, and Reference Data. While a plan will be established, it may be superseded through fast track support (see below). With the start of core, all non-core features will be removed. For simplicity, this release is referred to as 3.0.
4. **Establish separate Reference and UI Projects.** This may include defining a standard *deployment*

The focus of the incremental approach begins with a release that includes enhanced modularity support, followed by a workstream that spans across successive releases, and includes a build-out of core and additional architectural features. Advantages of frequent, incremental releases include both providing the community with regular opportunities to participate and provide feedback, as well as greater agility to respond to shifting priorities

In order to remain responsive to project needs, the OpenLMIS project team will enable “fast track support.” Fast track support gives priority to core features and extension points based on the needs of upcoming OpenLMIS deployments. If a planned project requires a core feature that is not in the short-term plan, the OpenLMIS project team can shift effort to it. This can include collaborative design, support and implementation work. Fast track support helps deliver on the short-term needs of a specific project while contributing to the overall re-architecture effort.

Fast track support will be granted only after a collaborative review with the deployment team. While every effort will be made to assist, the OpenLMIS team must balance available resources, timelines, and other architectural priorities. For example, fast track support may not be available if the required functionality cannot be delivered in time, it would divert from a critical architectural feature.

Figure 3 illustrates the general release plan.**

*Figure 3 Timeline release plan*

## Schedule (Incremental)

In September, the Technical committee drafted a list of milestones for 2016. Major milestones are detailed below, while the full list may be found [here](https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=30638284).

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.

| **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 | January 2016 |
| **2.0 stabilization Release** | Starting point for new projects  Improved data migration | March 2016 |
| Coding Standards/Contribution Guide, standards enforced | Community-enforced standards for updating master branch | March 2016 |
| Shared 2.0 Demo server | Public server for demos and reference | March 2016 |
| Modularity and Domain Model design | Functional Spec.  Classes for portion of model.  Beginning of Core API. | April 2016? |
| **Modularity Release** | Enhanced modularity | June 2016\*\* |
| Determine scope for next release | Select portion of core/extension points to implement | June 2016 |
| **3.0 Release** | First core/extension points; feature de-scope | TBD |
| Monthly updates and demos | Display progress, highlight critical decisions and changes | March through May 2017 |
| Release iterations | Continued architecture/infrastructure updates, based on remaining work and feedback from prior releases |  |

\*\* Dates at this point subject to community priorities, design estimates, etc.

## Complete Rewrite

The second approach is to simply create a completely new version of OpenLMIS.

The “minimum viable product” for a new application must match the functionality of OpenLMIS 2.0 (with global features toggled off).

The blank slate approach frees OpenLMIS from working within 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. Modularity and extensibility are included from the start.

The downside is that no value can be derived until the core functionality is complete. It is unlikely this can be achieved in 2016 with current resources. To best serve the community, sufficient funding is required to complete the rewrite within twelve months.

In terms of migrating OpenLMIS deployments to the rewritten version, a technical process would be defined for a data migration and the OpenLMIS tech team can provide guidance. The migrating deployment will need to invest time and resources into the planning and execution of a migration.

An emerging comparison list on the rewrite vs. incremental approaches is on the [OpenLMIS Wiki](https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=37912604).

# Not Included in 2016 Scope

Given the large scope of this project and current resources, the full re-architecture vision will likely not be achieved in 2016. Some items not scheduled for 2016 include:

* Entire OpenLMIS core feature set implemented as Core with extension points
* A complete reference UI or “easy” UI customization
* 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
* New feature support (separately funded projects may contribute additional features, so long as they conform to community-accepted standards, etc.)

# Resourcing

The OpenLMIS global software development team, housed at VillageReach and funded by the Bill & Melinda Gates Foundation OpenLMIS grant, will be fully allocated to this project over the duration of 2016. This includes three full time software developers, a half-time product manager/BA, and a software development manager dedicated at 80%.

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 detailed design and implementation. Major questions involving features, functionality or administration of OpenLMIS should involve the Product committee.

# Product Roadmap and Related Software Development Initiatives

In September, community members 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, meaning that other roadmap items will not be built out during this time period. Therefore expansion in the OpenLMIS feature footprint to be generated by the OpenLMIS global team in 2016 is not expected.

Individual implementation projects or other funded initiatives may still 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 projects should begin with the latest 2.0 release tag, 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, maintained in GitHub. Projects should not assume that their updates, especially changes made to forthcoming core logic, will be accepted back into the master.

Due to full allocation of VillageReach global team 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 continue to 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, and will provide periodic progress updates to the Product and Governance groups.

# Risks

| **Risk** | **Likelihood** | **Impact** | **Mitigation** |
| --- | --- | --- | --- |
| Lack of Community Involvement | Low | Decision-making is not representative; reduced staffing for work | * Several full-time VillageReach personnel dedicated to this effort for 2016 * A prioritized list of objectives helps 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 expertise 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 | Community available to help advise |
| Refactor overhead (adding new concepts in old code) shows incremental 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

# Appendix A – Core and Modules

|  |  |
| --- | --- |
| Core Code | * Highly stable and well-tested * Changes infrequently * Regular release schedule * Supported by the OpenLMIS Community (bug fixes, etc) |
| Community-Supported Module | * 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 |

# Appendix B – Re-architecture Options (not-prioritized)

## Option 1: Modularity Only

By using the existing gradle sub-project functionality the concept of adding functionality at build time in a project-specific repository will be formalized.

This is a critical inclusion, and modularity support is prioritized in the first few releases. Modularity will help greatly in resolving contribution hurdles, but does not resolve the extensibility problem. A separate module 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 is not an ideal situation; modularity is a critical piece, but is not the complete solution.

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

In this approach, core OpenLMIS functionality is updated 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.

The disadvantage of this approach is the work required to keep compatibility between layers, especially as that work may be discarded as each layer is updated. It is too restrictive, not permitting sensible change in other areas of the product.

## Option 4: 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, for example using a REST-style API for the client API. Operating within existing OpenLMIS code, core code will continue to be written in Java, using Postgres as the data store.

The 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 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.: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