3 STRATEGIES FOR DEVELOPING EFFICIENT LABVIEW PROJECTS USING A TEAM-BASED APPROACH

ERDOSMILLER

Introduction

“It takes one woman nine months to make a baby, but nine women can’t make a baby in one month.”

Adding more developers to a project doesn’t always increase efficiency. In fact, additional developers often create the “too many cooks in the kitchen” effect and decrease project efficiency. This effect is articulated by Brook’s Law, which states that “adding manpower to a late software project makes it later.” This phenomenon does not just occur with late projects. It can occur on any project because:

1. It takes time for people to ramp-up and contribute to a project in a productive manner
2. Communication efficiency tends to decrease as you add more people to a project
3. Tasks cannot be infinitely subdivided

Let us consider development using the National Instruments LabVIEW graphical programming language. While LabVIEW includes a variety of features that reduce development time compared to other programming languages, these efficiencies can be negated by adding development resources without an appropriate structure and plan. This is especially true of graphical programming languages because tracking and merging code changes pose unique challenges. Because of anticipated inefficiencies, companies often decide to use a single development resource for LabVIEW projects, causing a lengthy development time. However, proper LabVIEW-specific project management processes actually grant efficiencies and reduce time to market while minimizing the downside of having a team size larger than one.

Several obvious benefits can be gained by utilizing multiple development resources on LabVIEW projects. First, multiple developers, used correctly, can accelerate time to market. With proper structure and process, the gains are nonlinear. For example, it is possible to finish a project in six months with two developers that would normally take one developer two years. Second, multiple resources create redundancy that keeps a project on track when people must be shifted to new projects, an employee needs sick or vacation days, or a significant life event occurs for a team member. When done properly, the development company and customer ultimately save time and money and gain personnel redundancy.

With the right strategies in place, you can not only mitigate the risk of decreased efficiency, but sometimes maintain or even increase efficiency as resources are added. This paper discusses the following practical applications of project management to achieve these goals:

1. Select a predetermined workflow with defined roles and requirements
2. Use a LabVIEW architecture that avoids merging changes in source control
3. Develop and communicate a unified project life cycle

The suggestions in this paper are written from the viewpoint of a design services company developing products for a customer that will then manufacture and sell products to their customers (end users). Many references are specific to the National Instruments graphical programming language, LabVIEW.

Strategy #1: Select a Predetermined Workflow With Defined Roles and Requirements

To ensure your multi-developer project runs smoothly and that you achieve optimal efficiency, you need to effectively manage your requirements and clearly define the roles and responsibilities within your team. The two widest-known development strategies are waterfall and Agile (Figure 1).

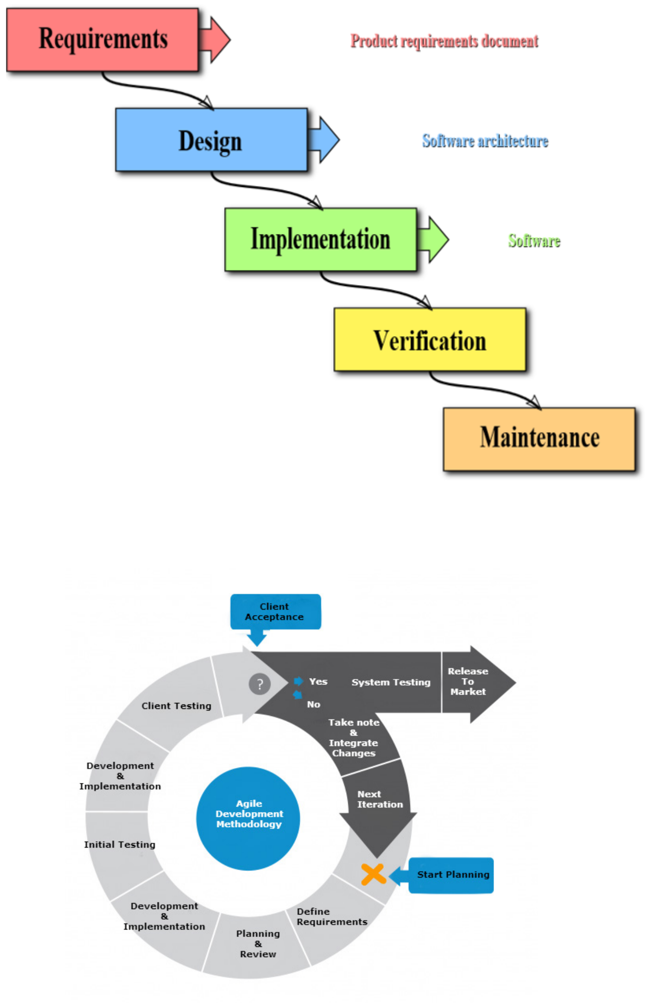


Figure 1. A Graphical Representations of the Waterfall and Agile Workflow Processes

The Waterfall Method

The waterfall model is based on the concept that each stage of a product's life cycle takes place in sequence so that progress flows steadily downward through each stage, like a waterfall (Figure 1). With this method, requirements must be clearly and thoroughly defined, which is time consuming and delays a project start. However, potential issues can be unveiled sooner in the design process if all requirements are well thought out and adequately discussed. Additionally, the waterfall method is a linear process that emphasizes documentation, which is reassuring to many organizations.

A drawback of this methodology is that requirements are static, and modifying requirements can cause you to start from the beginning, costing time and money. This is a challenge, especially when:

1. All customer decision-makers do not thoroughly review and agree to the initial requirements (there are always more people that want or need input if you ask)
2. The budget and schedule are locked in based on the initial requirements

Poorly defined waterfall projects tend to create friction between the design company and the customer. As requirements change, formal change orders are necessary to realign expectations, a process that often creates difficult conversations. The change order conversation is softened if expectations for scope additions are set at the beginning of the project.

Questions to ask include:

1. What is the target market for the product?
2. How will the customers use the product?
3. How will the user experience be validated and improved with target end-users?
4. How will end-users solicit feedback and request feature changes?
5. With which standards and regulations must the product comply? How will compliance be assured?
6. How will the product be tested as manufacturing volume increases?
7. What is the process to determine which changes get added?
8. How will the marketing team be involved in the product release?
9. Who else has input on this product before release to market?
10. What happens if this project goes over budget by 10, 25, or 50 percent?
11. What is the approval process when new features are identified and agreed to?

When a change is requested, a helpful strategy is to advise project managers to offer a substitution of new features for original features that have not yet been implemented. A feature swap can often allow the project to keep its original cost and schedule by removing features that are no longer as important as originally estimated.

The Agile Method

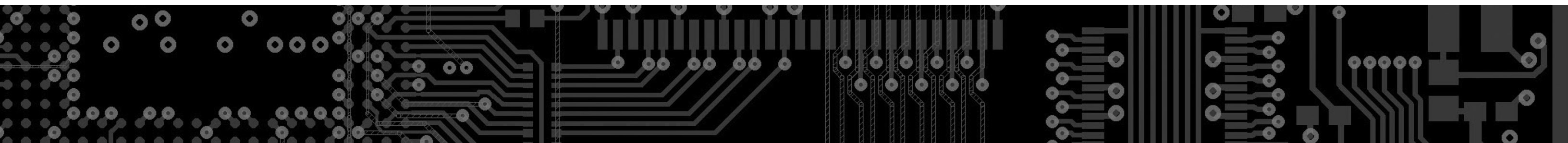
The Agile method uses an incremental, iterative approach, where cross-functional teams work on successive versions of the product until development is stopped or the customer has captured the target market to their satisfaction. Using this method, working software is delivered more quickly but with fewer initial features. Thise approach requires continuous improvement and feature changes—a process that embraces requirements changing over time.

It is particularly difficult to pursue Agile development when you are designing a product for an external customer. Because of Agile's emphasis on the release of minimum viable products (MVPs) and incremental feature additions, customers often have negative perceptions of early iterations if they do not expect an MVP. Some examples of common complaints with an Agile product include:

1. Why is the product incomplete/where is feature X?
2. Why is the product unpolished?
3. Where is the documentation?
4. How much will it cost to get to full commercialization?

Additionally, it is more difficult to implement a fixed-price project model using Agile, leaving many design companies to default to time-and-materials contracts.

Like the waterfall method, customer communication is critical to avoid mismatched expectations. Separating the project into distinct smaller phases is one successful strategy to keep customer expectations in line and keep



the development team and customer accountable to deliver and accept what is actually needed and expected. A phased approach also makes fixed-price contracts more realistic with Agile development, which rewards the development company for over performing.

Develop Concise, Specific, and Unique Requirements

Every large application needs a method to organize functionality and dictate the method by which that functionality is accessed. Regardless of the workflow you choose, it is important to develop concise, specific, and unique requirements for your software. At the most basic level, your requirements need to state what you want the software to do and not how you want the software to do it. Requirements also need to be single items and not a combination of separate functions. For example, you would not want to state that the "software can control the pump manually and automatically" because these are two compete different functions.

Additionally, requirements should be divided into phases, with phase 1 having just enough functionality to create the MVP—a lean version of the product that is functional for the intended purpose but not yet be feature rich. As an example, a bicycle would be an MVP when designing a car, but a car chassis with four wheels and nothing else would not be an MVP. Your requirements document should also identify the scope of the project, which includes:

1. Goals
2. Milestones
3. Expected costs
4. Deliverables
5. Acceptance criteria
6. Change management process

A change management process is part of the requirements document and defines how the client requests scope additions from added, changed, or even unanticipated features required for complete functionality (more details in strategy 3). In the end, your requirements document should serve as a written agreement between your organization and the client detailing the final application. It is critical that this document is clearly communicated to all of the project's stakeholders.

For LabVIEW applications, an architecture should be defined in the requirements document. Every sufficiently large application needs a method to organize functionality and dictate how that functionality is accessed. There are a variety of LabVIEW architectures in use today, but we recommend either a Model/View/View-Model or Atomic architecture. Further details are provided about these two approaches in strategy 2.

Clearly Define the Roles and Responsibilities of Your Project Team

A proper project plan defines the roles of the project team. Figure 2 outlines what a project team could look like for a services company that develops products for a customer.

The groups outlined in Figure 2 can be defined in the following manner:

- **Core team**—Resources that are fully dedicated to the project from start to finish
- **Transient Internal**—An internal resource that is not fully dedicated to the project but may be called upon if needed
- **External Resource**—Any resources that are not employees of the development company

When identifying your core team, you should ensure the team members have the proper amount of

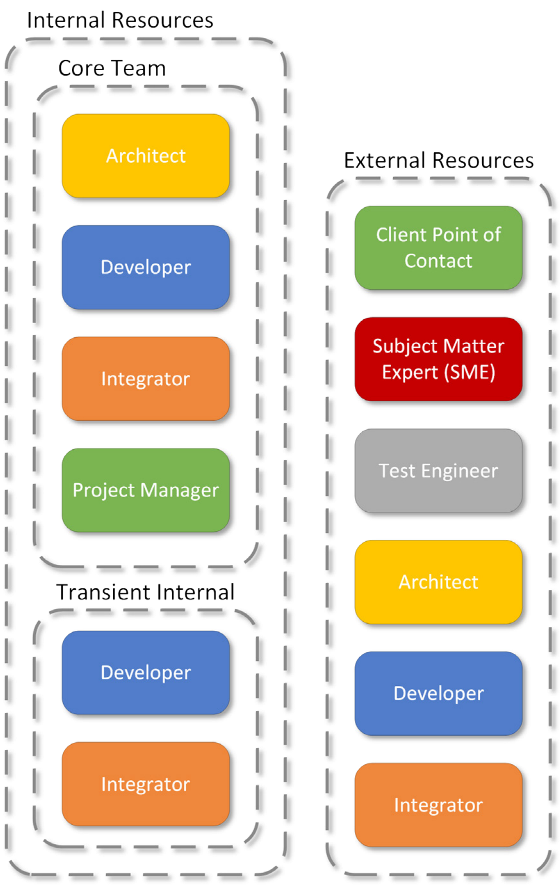


Figure 2. A Model of a Software Development Team

experience—Certified LabVIEW Associate Developers (CLADs), Certified LabVIEW Developers (CLDs), and Certified LabVIEW Architects (CLAs). These are National Instruments certification that designate experience and expertise. Each resource level offers widely different experience and should be utilized properly. It may seem that assigning a fresh CLAD to a 500-hour industrial controls project will save money and increase profit, but usually this results in schedule delays, application-breaking bugs, and upset customers. The development responsibilities should be divided among the core team in the following manner:

a) Developer (CLAD/CLD)–Scope of work is narrow and task oriented.

b) Integrator (CLD/CLA)—May take modules or pieces of code written by developers to ensure functionality happens in a specific order.

c) Architect (CLA)—Drafts the big picture of the project and is generally heavily involved at the start and the release of a project. Throughout the project, his or her job is to ensure the software maintains its intended design principles and functions within the context of its requirements.

Using this approach and assigning responsibilities within a team-based environment establishes a chain of command.

Two key business roles should be established for each project—a project manager and a client manager (for smaller projects these could be the same person). As the name implies, the project manager focuses on the project and has the following duties:

1. Be the decision maker on the project and be seen by the customer as such
2. Serve as the primary point of contact for the customer
3. Help the customer understand the business impact of project decisions
4. Assign resources, tasks, and deadlines
5. Hold the team accountable to task assignments
6. Take responsibility for the success or failure of the project

The client manager is a highly technical salesperson who:

1. Is responsible for the long-term customer relationship
2. Co-develops the strategic project vision with the customer
3. Helps the customer make product line roadmap decisions with input from the technical team

On some projects, it may become necessary to bring in an external resource to serve as a subject matter expert, test engineer, or even part of your core development team. There are a variety of ways you can ensure that external resources are well -utilized. First, determine each team member's preferred communication method. If the contractor prefers in-person communication but is geographically disparate, make a plan to address this through frequent, in-person meetings or working sessions. Similarly establish a "teaching" plan to ensure external resources can get up-to-speed on a project as quickly as possible.

It is also important to ensure all external resources have proper system access, especially to necessary proprietary information. If there are concerns over the reuse or sharing of any of this type of information, have your external resources sign a non-disclosure agreement. It is common for standard service agreements to limit information disclosure to third-party contractors, so validate with your lawyer that this is acceptable before finalizing your project plan.

One unique benefit that an external development resource may offer is they may have previously worked on a similar application from which they may have the ability to reuse code. This allows the developer to spend more time working on new features instead of reinventing the wheel. However, it is important to vet that this intellectual property is owned or licensed and free and clear for sublicensing before considering it for customer projects. Another point often overlooked is insurance for contractors. You should ensure that your liability policy covers contractors, and if it does not, change your policy or require that contractors carry their own business insurance. Contractors can be a powerful augmentation to your team, but you must take some additional steps to ensure seamless integration with the team.

Strategy #2: Use a LabVIEW Architecture That Avoids Merging Changes in Source Control

Choosing the right LabVIEW architecture is critical to efficiently collaborating with multiple developers and effectively managing source code. This section outlines best practices for:

1. Developing an effective LabVIEW architecture
2. Organizing your modules to avoid time-consuming rework
3. Simplifying merging and managing source control conflicts

Characteristics of a Good LabVIEW Architecture

Although LabVIEW provides an easy-to-use application program interface (API) for common tasks, LabVIEW is still a programming language, which means that as the need for more integrated functionality increases, so does the need for a defined architecture. Architectures serve two main purposes. First, they organize your functionality, and second, they provide the ability for your organized functionality to work together. A good architecture makes it easier to achieve the functionality outlined in a project's requirements document and is defined by the following characteristics:

- **Modularity**—Features are easy to add in future phases
- **Low coupling/high cohesion**—Modules work well together (high cohesion), but should not be so integrated (low coupling) that it becomes hard to modify one without modifying the other
- **Scalability**—A scalable application requires minimal changes to make a feature change
- **Ability to merge changes via discard in source control**—It should be easy to determine which changes to keep in source control and which to discard

As we mentioned, two recommended architectures are the Model/View/View-Model (MVVM) and Atomic architectures. Let us take a close look at each of these.

Model/View/View-Model Architecture Overview

The MVVM architecture consists of three major components:

- **Model**—The data model representing all the private and public data for each module
- **View-Model**—This mode is responsible for providing views with the latest data stored in the model as well as updating the data model in response to external actions
- **View**—A user interface for displaying the model and providing input to the model

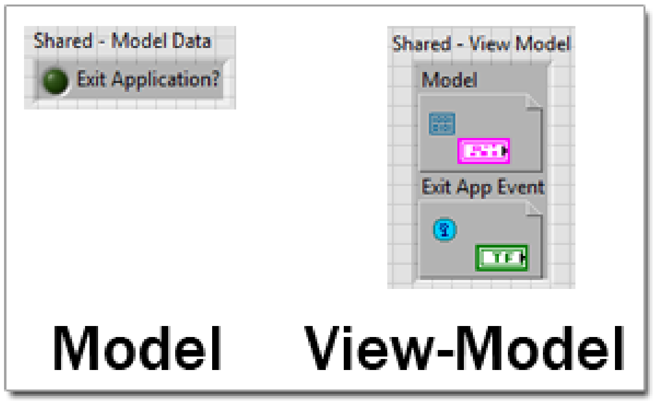


Figure 3. LabVIEW Representation of the Model and View-Model

The MVVM architecture implements multiple asynchronous modules that maintain their own data model in the form of a data value reference (DVR) as well as their own view-model in the form of user events for data binding. Every module's view-model is then combined into a single cluster and this cluster is provided to every module and sub-VI within the application. This grants global access to data binding events and model data. Through the use of asynchronous modules, functionality can easily be divided among team members with little interference and dependency, which makes this architecture excel in a team-based environment. In addition, the modular approach makes source control management much easier and creates less merge conflicts. Some disadvantages of this approach include:

- Difficulty implementing synchronous operation between modules when two modules need to run specific actions in a specific order to achieve proper functionality
- All model data and event references are available globally

When working with MVVM in a team environment, it is best to divide duties by module. Once modules are assigned to developers, the lead architect has a choice on how to begin the initial project. The first option is for the lead architect to generate the project, folder structure, and an empty type-defined cluster for holding view-models. The second option is for the lead architect to generate the project, folder structure, and the initial type definitions for all module's model data and view-models as well as the cluster of view -models to be used as subVI inputs.

Atomic Architecture Overview

Another architecture we recommend is the Atomic architecture (Figure 4). This structure is based on the idea that a project can be split into distinct unrelated modules that are integrated via a main VI. This architecture works well in multi-developer environments because the source code for each module is self-contained and modular, which means VIs can be developed concurrently without creating merge conflicts. Aside from running the main VI for each module, the only way modules are accessed is through the API, which creates separation between functionality and how that functionality is accessed. Interdependent code does not interact directly with the functionality of other modules, so it is very easy to fix bugs and make updates as long as the API does not change.

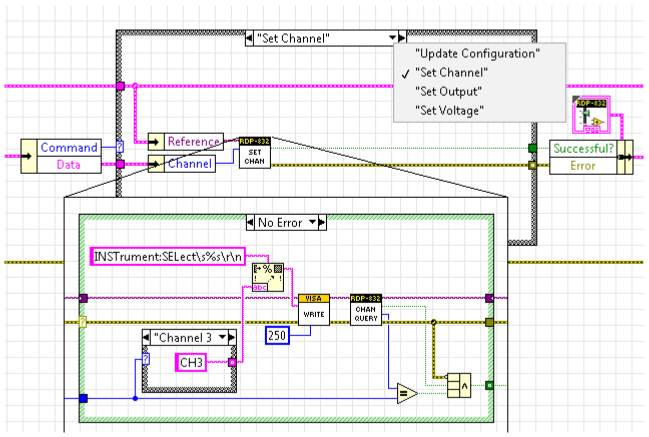


Figure 4. A Block Diagram of an Atomic Architecture in LabVIEW

One of the greatest advantages of an Atomic architecture is that modules are highly portable—in many cases, the code can be copied and pasted into the module directory of the new project. Surprisingly, it is often sufficient to copy module directories of multiple interdependent modules into the new project and resolve dependencies (if any).

By design, the Atomic architecture heavily depends on higher-level VIs to access the functionality inside each module, handle errors and provide high-level verification that a process was successfully completed. By using an Atomic architecture, an integrator or architect may need significantly more time to integrate module functionality into a Main VI or user interface, which would increase the total cost of the project.

Simplifying Merging and Managing Source Control Conflicts

Source control is a term used to describe the backup method used for source code. Similar to the differences of LabVIEW compared to its text-based analogues, a proper source control solution for LabVIEW can be unique and complex, especially when working with simultaneous developers. Source code that is backed up allows for data recovery in the event of a natural disaster, hardware failure, or theft, and has the added benefit of consolidating source code from multiple developers in one easily accessible location. Some common methods of source control include the following:

- **Local**—Stored on the same machine developing the code in an alternate folder or archive (e.g. *.zip, *.tar, *tar.gz)
- **Client-Server**—Stored on an alternate computer not being used to develop the code itself
- **Peer-to-Peer (Distributed)—**Stored both locally and on a server

Each method has benefits and drawbacks, but source control complexity tends to increase proportionally with the amount of desired functionality. Some advanced features accessible with source control include:

- Concurrent access to source code
- Change roll-back
- Bug fixes for current releases
- Development/feature add-ons for future releases

Using source control with LabVIEW is not straightforward. LabVIEW stores its source code in a binary format that is not human readable, which means traditional diff/merge tools cannot be used. LabVIEW is a compiled language and it will continuously re-compile a VI or control whose source has been modified, and it will also re-compile any VIs that call the control or VI. Thus, by editing one file in a LabVIEW architecture, any other files that call that file will also be recompiled and source control will assume multiple files were changed when only one file was changed. To determine which LabVIEW files were actually changed, it is recommended that the lead architect ensure all VIs are configured to separate compile code from source code.

While diff/merge tools exist for LabVIEW, they do not scale well in terms of time and effort required to resolve conflicts. In a large LabVIEW project, the best way to handle conflicts is to merge by discarding or overwriting changes. The s If each developer is aware of what they are responsible for, they can discard appropriate changes to files or folders. However, since this is a manual processIn projects where there is more than one developer working concurrently on the source code, there is a high chanceprobability of a mistake being mades, especially when there is more than one developer working concurrently on the source code.

Both MVVM and Atomic architectures help simplify source control and merging because of their modularity. Source control management best practices with these architectures include:

Creating one master repository with several sub-repositories using a source control solution such as Git. This greatly segregates each developer and can prevent write access to modules not belonging to that developer (Figure 5).

Following a branching methodology such as Git Flow, where every module can be represented as a feature branch and merged into the development branch at regular intervals (Figure 5).

“Merging” by discarding inappropriate changes and uploading appropriate changes associated with each corresponding module to source control.

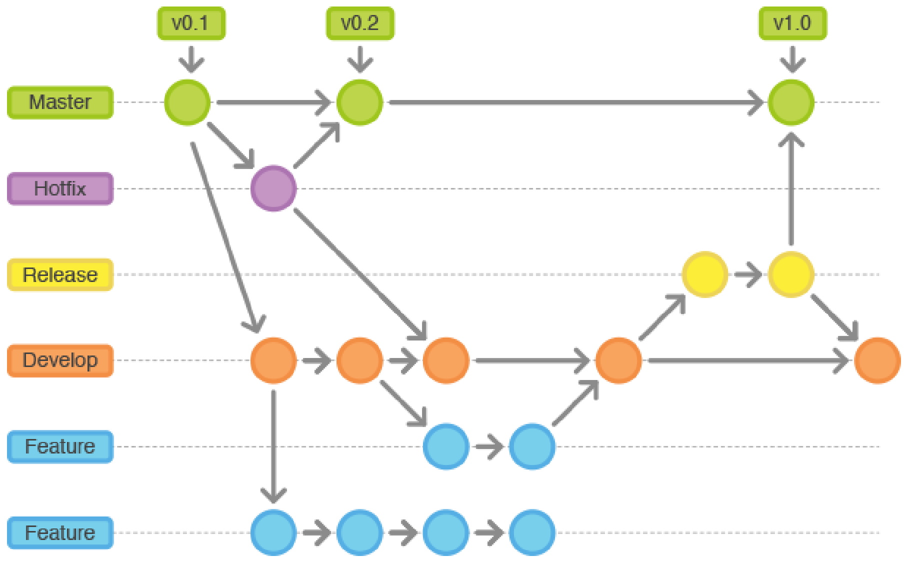


Figure 5. An Example of a Git Workflow

Strategy #3: Develop and Communicate a Unified Project Life Cycle

It is important to use a predetermined workflow for a project, but it is also critical to develop an effective project management approach that uses a clearly communicated project life cycle. Especially in multi-developer environments, clear requirements documents should be used to reduce unnecessary communication, minimize miscommunication, and help keep software development on-track. The project life cycle typically includes the following stages:

1. Understand Pain
2. Generate Requirements
3. Generate Test Scenario
4. Develop Proposal
5. Accept Requirements (customer)
6. Develop Software
7. Test Software
8. Release Software
9. Feature adds/bug fixes
10. Project is competed

It is important to note that during the project life cycle, there may be multiple iterations of steps 6 through 9 as features are tested and bugs are fixed. Once the project is complete, the development team delivers the final, agreed-upon deliverables.

Once software is sufficiently developed, the test scenarios established earlier in the project can be used to validate functionality. This can be done at the end of the project to determine if the project is complete or at an early stage of the project to determine if a specific implementation is successful. When working with an external company for development, that company will validate whether or not the requirements that were agreed upon were met with the developed software. Typically, the third-party company does not have the metrics available to verify whether or not the software application will meet the business use case for the client and be successful from that perspective. For example, an external developer can verify that a PID can maintain a temperature setpoint within ±2° but they cannot verify whether or not a new product will cause a 30 percent increase in overall revenue by updating the technology in an existing product line.

Software validation can generally occur at any time during or after development. Oftentimes, software validation is the only way to determine whether or not an implementation is successful or whether or not a requirement was met. When software validation fails, it often means that a specific implementation does not work, but this can also have the effect of requiring a change in scope. Scope changes can occur for a variety of reasons, including the following common reasons:

- Time and materials estimates based on specific implementation that was disproven
- Client wants to add a feature that is out-of-scope as defined by the requirements document
- Requirement was initially incorrect and has to be modified to produce the desired functionality.

Scope changes must be managed using a process where changes proposed by any party are discussed, mutually agreed upon, and authorized. Scope changes may dictate modifications to the requirements document and test scenarios. The change management process can help keep the project focused and under budget as well as keeping all involved parties informed and content even when making changes.

Once software development reaches a point where the lead developer is confident that the software will work as expected after testing in-house, an official round of software validation will occur by deploying a release version of the software in the environment it was designed to run in. This is called an site acceptance test (SAT). Release versions of software are commonly distributed using the following methods:

- **Installer/Updater—**An application that will install the LabVIEW application as well as install any dependencies
- **VI Package (*.vip)**—A single file that uses the VI Package Manager to install a version of the source distribution to a specific version of LabVIEW

Once this release version is officially validated, the project may be complete. Depending on the terms of the original proposal, the client may enter a warranty period where bugs that are in-scope can be resolved or a new development phase can be started where features are added.

Achieving Maximum Efficiency Using a Team-Based Approach

By using a predetermined workflow with defined roles and requirements, implementing a LabVIEW architecture that avoids merging changes in source control, and developing a coherent, unified communication strategy, your organization can work more effectively on applications that take advantage of multiple developers. In the end, by implementing these strategies, you can increase efficiency and decrease development time with your medium-to-large LabVIEW projects, even when adding more resources.

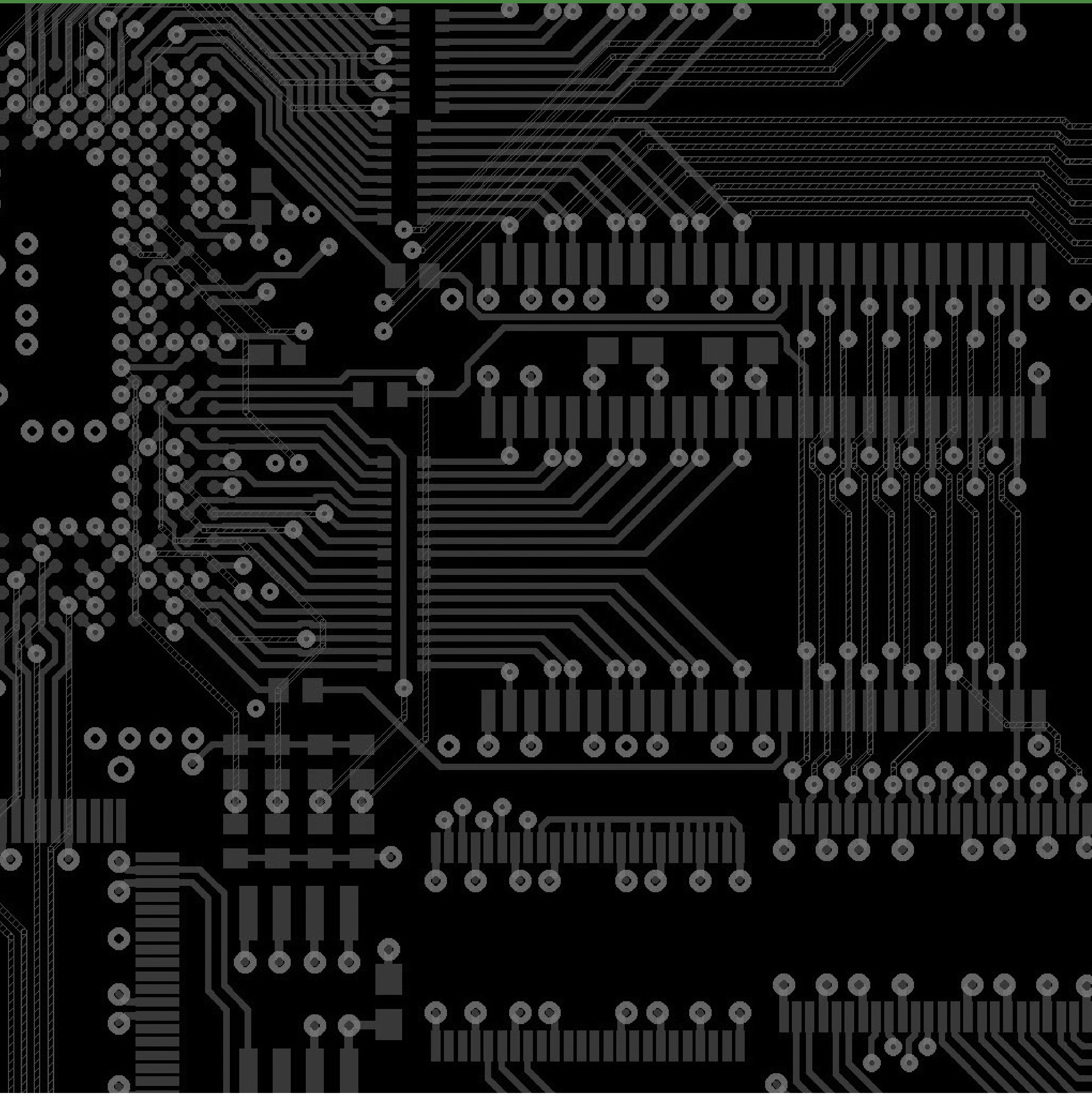
Learn more about Erdos Miller's Project Best Practices

Address

15120 Northwest Fwy.
Ste. 100
Houston, TX 77040

Contact

1-888-337-0869
info@erdosmiller.com
erdosmiller.com



ERDOSMILLER