Leading a Software Architecture Revolution

Part 2b: Tactical Prioritization

Marden Neubert, PagSeguro Ltd,—Brazil
Joseph W. Yoder, The Refactory, Inc,—USA

As a software system grows in complexity, technology evolves, and organizations seek new opportunities, software architecture can become unsuitable for the problems it should help solve. This scenario calls for a software architecture revolution—an extensive and profound transformation of a system's core structures to align it with current and future requirements. This process is challenging and demands significant organizational resources, commitment, and effective guidance. While there is extensive research on the technical aspects of architectural changes, leadership dynamics in such initiatives are only sparsely discussed. Leaders play a pivotal role in this journey, advocating for the revolution, deciding priorities, negotiating resource allocation, assessing progress, implementing corrective actions, and showcasing achievements. Our previous work introduced a pattern language for leading software architecture revolutions. This paper expands that language, presenting a set of tactical prioritization patterns drawn from real-world experiences to enhance the effectiveness of architecture revolution initiatives.

Categories and Subject Descriptors
• Software and its engineering ~ Agile software development • Social and professional topics ~ Software architectures; Software evolution; Design patterns; Agile software development

General Terms
Architecture, Management, Sustainable Delivery, Leadership, System Modernization, Architecture Revolution

Additional Keywords and Phrases
Software Development, Maintaining software, Evolutionary Architecture, Patternd, Pattern Sequences & Scenarios

ACM Reference Format:
Neubert, M., Yoder, J.W., "Leading a Software Architecture Revolution - Part 2b: Tactical Prioritization". HILLSIDE Proc. of 30th Pattern Lang. of Prog. (October), 41 pages.

Author's email address: marden.neubert@gmail.com, joe@refactory.com

1. Introduction

The software architecture of a system can be defined as the set of structures needed to reason about the system [Bass et al.]. These structures are composed of software elements, the relations among them, and the properties of both. The architecture manifests the design decisions related to the overall system structure and behavior.

Over time, however, as technology evolves, requirements change, and software complexity increases, the original architecture may become inadequate and an obstacle to further growth and development. This situation could be the result of forces such as the accumulation of technical debt, the introduction of new requirements incompatible with the existing architecture, changes in the organizational structure of the company maintaining the system, or the shift in market dynamics and business models.

In such a context, a gradual, evolutionary approach to altering the architecture might not be enough or even feasible. A more radical transformation, which we term a "software architecture revolution," is necessary. This endeavor requires a committed, organization-wide effort to drastically reshape the software architecture.

An architectural revolution is a radical change initiative. It affects not only the IT department but also ripples throughout the entire organization, impacting priorities, demanding considerable investment in time and resources, and requiring adaptation to the new architectural landscape. As such, clear communication and understanding of the revolution's motivations are crucial among all stakeholders involved in product development.

In previous works, we introduced patterns for helping leaders of architectural revolutions, organized in groups focused on "Creating Awareness," "Preparing and Measuring" [Neubert, Yoder 2022], and "Strategic Prioritization" [Neubert, Yoder 2023]. This paper expands that language by introducing patterns for "Tactical Prioritization," focused on assisting leaders and teams in prioritizing activities for the revolution initiative. It draws from practical experiences from the authors—a CTO who led a significant transformation and a consultant advising many companies in similar situations—to provide leaders of architectural revolutions with strategies for prioritizing their teams' activities.

The paper gives some background information, followed by an overview of the patterns. This is followed by some of the patterns for prioritizing activities when leading a revolution initiative. Brief descriptions (patlets) of the patterns are provided in **Appendix A**, which includes a summary of all patterns we have outlined in this language.

The patterns presented here are intended primarily for potential leaders of system modernization [Seacord et al.] initiatives, such as CTOs, CIOs, or Directors of Engineering. They may also prove beneficial to others involved with software architecture, including architects, developers, technical leaders, and even non-IT personnel.

In presenting these patterns, we adopt a modified Alexander style [Alexander et al.], wherein the context of the pattern precedes the first bold problem statement. This problem statement is followed by a discussion of the forces at play, the core of the solution, and then further details. Each pattern concludes with a discussion of consequences and related patterns (in bold).

2. Background

We use the term *software architecture revolution* to indicate that architectural work will be performed and managed as a dedicated initiative, instead of just organically, with tasks living side-by-side with new features in the teams' backlogs. We intend to contrast the name with "evolution" to highlight that the transformation process is perceptible to the company. As the dictionary definition clarifies, a revolution can be sudden, but it can also happen progressively.

When you need to make radical changes to a system's architecture, it is important to start by **Creating Awareness** of the issues so that you will have buy-in and support throughout the organization. Once you get the commitment, you should begin **Preparing and Measuring** to see whether you are moving in the right direction. **Figure 1** shows the patterns in these two groups, previously described in [Neubert, Yoder 2022].

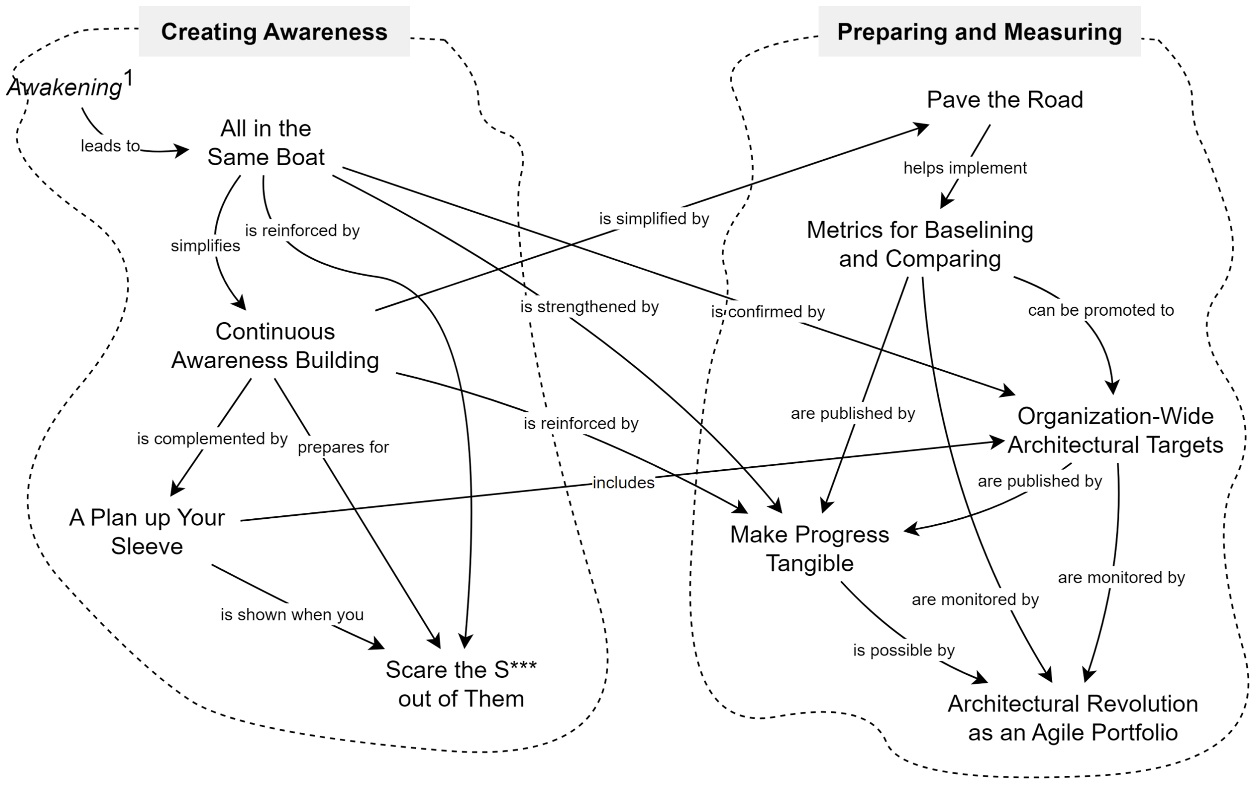


Figure 1: Creating Awareness and Preparing and Measuring Patterns

Before an architectural revolution can happen, the organization must become aware of the problems with its software and acknowledge it needs to be fixed. This begins with some form of ***Awakening*[1]**. Many things can lead to an ***Awakening***. It usually begins when an individual or a group (possibly teams) becomes aware that there are some issues or problems with the current system. This does not happen overnight and can start with a feeling that something is wrong. This can lead to some teams working to do a more extensive revision in order to "straighten things out." Although they make some progress, the more they do, the more problems they see,

[1] ***Awakening*** has not yet been written as a pattern, but could be. It is one of the first things that must happen before any architectural revolution takes place.

until ultimately it becomes clear that the only way they will be able to fix the architecture is with a radical change. After an ***Awakening***, the organization will understand the situation much more easily if the departments involved in product development are **All in The Same Boat**. **Continuous Awareness Building** can help IT teams and related areas sustain development and continue to promote the need for the new architecture. You should have **A Plan up Your Sleeve** to offer some hope to the teams and pitch it to get approval for the revolution initiative. When you have the opportunity to talk to upper management about the initiative, be emphatic and **Scare the S*** Out of Them** with the risks presented by the current architecture.

Once you have started, it is good to **Pave the Road** by creating the infrastructure and educating the organization about the new architecture. As you start the revolution initiative, it is helpful to collect **Metrics for Baselining and Comparing** the architecture before and after each step. These metrics can serve as a basis for defining **Organization-Wide Architectural Targets**, which is a strong message to the organization that the revolution is a priority. As the initiative evolves, keep in mind that it is not easy to have a clear feeling of the progress in a journey of changing software architecture. **Make Progress Tangible** so that teams and business leaders will be motivated by the advances. As a best practice to have insight and control over the initiative, we recommend that you manage **Architectural Revolution as an Agile Portfolio**.

The next step is to ensure that teams are working on the most important activities in the revolution, which is done by **Prioritizing Activities**. This is essential because you want to deliver value as soon as possible. While value here does not mean a new feature, customers will benefit from a more stable service, fewer defects, and more frequent releases. Internal stakeholders will also gain from internal characteristics, such as more flexibility, better testing, more straightforward deployment, and better tooling.

There are strategic decisions that need to be made when you start **Prioritizing Activities**, which lead to a broader impact in the shape of the revolution initiative [Neubert, Yoder 2023]. For example, these decisions include how the new architecture will be rolled out and whether new feature development will be halted during the revolution, as shown in **Figure 2**.

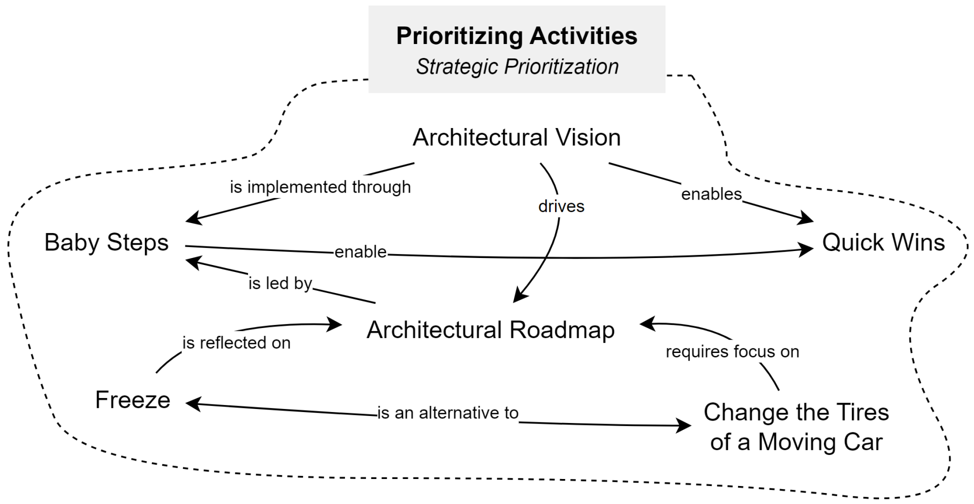


Figure 2: Strategic Prioritization Patterns

Early on your revolution journey, you need to decide on strategic priorities and guidelines. These decisions will have a broader impact in the initiative because they will define long-term goals and overarching approaches to the challenges you will face. These foundational decisions start with an **Architectural Vision** that inspires teams to join the initiative and guides them toward the desired goal. This vision can be used to create an **Architectural Roadmap** which is a rough plan for when various architectural features should be addressed during the revolution. You should make progress toward the new architecture by taking **Baby Steps** and creating just enough infrastructure to support simple scenarios and validate the vision. As you start migrating to the new architecture, look for **Quick Wins** so that teams can learn fast and be motivated by delivering some results. A frequent discussion in prioritization is whether you should **Freeze** all other initiatives and focus solely on the revolution to mitigate the risks sooner and reap the benefits of the new architecture. The alternative is to **Change the Tires of a Moving Car** and keep other initiatives going. These two options can coexist: you may choose to freeze some parts of the system to migrate them while other development initiatives are allowed to move on.

This paper will focus on the prioritization decisions after the strategic definitions, which we call tactical prioritization. The patterns discussed here are equally important because they offer practical guidance to help leaders and teams prioritize architectural tasks during the revolution initiative. The patterns presented in this paper are shown in the tactical section of **Figure 3**.

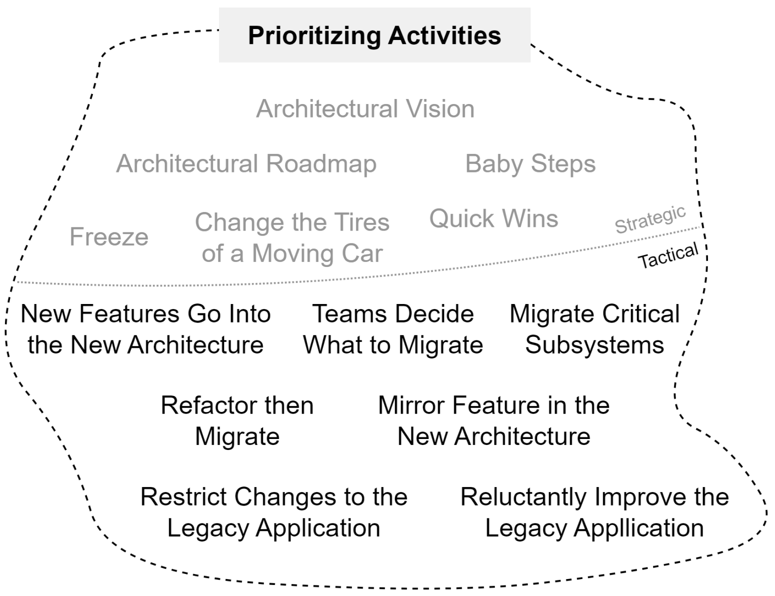

Prioritizing Activities
Architectural Vision
Architectural Roadmap
Baby Steps
Freeze
Change the Tires of a Moving Car
Quick Wins
Strategic
Tactical
New Features Go Into the New Architecture
Teams Decide What to Migrate
Migrate Critical Subsystems
Refactor then Migrate
Mirror Feature in the New Architecture
Restrict Changes to the Legacy Application
Reluctantly Improve the Legacy Appllication

Figure 3: Tactical Prioritization Patterns

A key prioritization tactic is to ensure that teams do not create more backlog for themselves by making **New Features Go Into the New Architecture**, even if it is necessary to migrate a larger subsystem to add a small feature. It is valuable to have **Teams Decide What to Migrate** because they know best what affects their work. Prioritize the improvement of critical and risky parts of the architecture by having teams **Migrate Critical Subsystems**. Sometimes the legacy application code is so poor that it pays off to **Refactor then Migrate** a subsystem. When a certain subsystem is too complex and risky to be migrated without disruption, or using the new version would require some action from external parties (for instance, customers or partners), **Mirror Feature in the New Architecture** and keep both versions working in parallel. Finally, create gatekeepers to **Restrict Changes to the Legacy Application** and allow only top priority initiatives to be implemented in the old architecture and **Reluctantly Improve the Legacy Application** to avoid increasing risks.

3. New Features Go Into the New Architecture

Photo by Steve Lieman on Unsplash

Your organization relies on a system that has become increasingly difficult to work with. You had an ***Awakening*** and realized that there is a mismatch between the current architecture and the organization's needs. You have concluded that you must take action to change this situation and have started an architectural revolution.

You have been making progress toward a new architecture, deploying and testing an **Architectural Vision** with **Baby Steps**. Some teams may have already created software in the new architecture, achieving **Quick Wins** and making you more confident that the vision is viable.

In the meantime, business leaders have continued to request new features and improvements on existing ones, often stressing that these are urgent demands. Despite your efforts to **Pave the Road**, some teams may not yet feel confident developing in the new architecture and believe that it would be faster to work on the legacy system.

How do you create and evolve features in the context of an architectural revolution so that the organization can avoid burdening the legacy system?

Once you have started an architectural revolution and demonstrated the risks of the legacy system via **Continuous Awareness Building**, you do not want the initiative's backlog to grow because teams keep creating new features in the old architecture. This situation would send a wrong message to the organization, suggesting that using the new architecture is optional and the revolution initiative should not be taken seriously. That said, ideally, all efforts toward developing features and evolving existing ones should be directed to the new architecture.

However, in the early stages of the revolution initiative, teams might take longer to develop features in the new architecture. One reason is that the legacy system already contains all the logic currently making up their solutions. Making a small change to an existing feature may be simple in the legacy system but would require a significant migration before it can be implemented in the new architecture. Another explanation is that teams will usually produce more conservative estimates when they do not fully understand the technical tasks that make up an activity [Cohn]. Given that some teams may not yet be proficient in the new architecture soon after its rollout, estimates could be higher at that point.

Business peers may pressure teams to use the legacy system to get features delivered faster. This phenomenon might happen even after you get **All in the Same Boat** and create awareness about the issues with the old architecture. One possible explanation is our tendency to prioritize urgent topics over less important ones, such as long-term strategic goals [Kerzner]. The push for implementing features in the legacy system comes from those working closer to the teams, such as Product Managers and Product Owners, driven by individual goals.

Nonetheless, some change requests may indeed be critical and negatively impact the business if not carried out as soon as possible. For instance, minor feature improvements, bug fixes, and security patches would likely take much longer to implement in the new architecture because they would require the migration of components or whole subsystems. The delay caused by forcing teams to implement them in the new architecture could lead to a decline in user satisfaction, loss of customers, and increased risks.

Therefore, mandate that teams develop and change features only in the new architecture.

Communicate clearly to the whole organization that development from now on should happen exclusively in the new architecture. Refer to the information you share while applying **Continuous Awareness Building** to explain the risks of continuing to use the legacy system, especially when creating new features. Leverage **All in the Same Boat** to show that, although switching to the new architecture can lead to some early delays, teams will eventually perform better, and users will be better served. Use initial results from **Metrics for Baselining and Comparing** to support your claims. Secure the support from top business leaders—they will be vital in validating the message within their hierarchy. If necessary, **Scare the S*** Out of Them**.

Define a simple and transparent process for reviewing proposed changes to the legacy system. Acknowledge that some requests—especially those tiny, incremental changes—will require careful consideration when deciding whether to implement them in the old architecture. Show some flexibility so that you are considered reasonable, but do not concede to decisions that will harm the future of the revolution initiative. Select a group of people, combining representatives from both IT and business and including both leaders and specialists, to decide which demands should be implemented in the legacy system. For more details on such a process, refer to **Restrict Changes to the Legacy Application**.

Provide continuous support to the teams developing features in or migrating to the new architecture. Offer training, create documentation, and give them tools so they will be as effective as possible. Remember to continually **Pave the Road** as you discover new ways to help teams engage with the revolution initiative. Motivate teams by showing them **Quick Wins** already achieved—this will also make them more confident that the new architecture is reliable.

Monitor the legacy system to ensure that changes are not creeping in. This situation could happen because some teams may accidentally use the old architecture or intentionally disrespect the mandate. Use **Metrics for Baselining and Comparing** to identify these situations and consider establishing a policy to avoid them altogether, such as allowing only reviewed changes into the legacy codebase.

✻ ✻ ✻

The main benefit of **New Features Go Into the New Architecture** is protecting the revolution initiative by preventing its backlog from growing. It also switches the organization's mindset to start using the new architecture and shows that the revolution initiative is here to stay. When you carefully consider which changes you will permit in the legacy system, you also reinforce the feeling of being **All in the Same Boat**.

New Features Go Into the New Architecture helps you avoid unnecessary or untimely changes during the architectural revolution. It helps redirect migration efforts toward parts of the system that are being actively evolved and are more likely to be relevant to the business. Consequently, no time or energy is invested in migrating features that do not change frequently, which would be less beneficial to the organization.

However, having **New Features Go Into the New Architecture** is not enough to guide the prioritization of the migration tasks in the revolution initiative. Migrating parts of the legacy system in the order that the change requests are arriving may not be an ideal strategy for the migration tasks. There can be another downside to relying solely on this pattern: not prioritizing for migration those parts of the system that do not change frequently but are critical to the organization. These parts can include core subsystems that are rarely modified but pose risks if they remain in the legacy system.

Another consequence of **New Features Go Into the New Architecture** is that it leads to more decoupled architectures. When teams have to use the new architecture for creating or changing features, they either build a small decoupled component in the new architecture to support the new feature or migrate minimal logic before implementing the change. This consequence is positive when you come from a highly coupled architecture and suffer from its negative impacts. Conversely, a solution that is too sparse can lead to other problems, such as high complexity, expensive infrastructure, and poor performance.

One possible downside of this pattern is having delays in new or improved features because they were estimated for development in the legacy system. If the teams responsible for those features are less comfortable working with the new architecture, or if they have to migrate a significant part of the legacy system before implementing the change, they will take longer to deliver. This situation can lead to another negative effect: business people who work directly with teams can feel incentivized to stress the urgency of their demands or inflate their benefits to get them implemented faster in the legacy system. To mitigate that, it is vital to have the buy-in from the leadership team and an effective review process via **Restrict Changes to the Legacy Application**.

The review process for allowing changes to the legacy system can also be a source of issues with **New Features Go Into the New Architecture**. If the criteria are too strict, it will lead to significant delays in small feature requests and bug fixes, causing business leaders to be discontent with the revolution initiative. If, on the other hand, the evaluation is permissive, it will defeat the purpose of the pattern and increase the complexity of the legacy system.

Related Patterns

Getting business leaders and top executives **All in The Same Boat** is crucial to establishing the mandate that **New Features Go Into the New Architecture**. You need their support when plans for new features have to be adjusted to account for higher estimates.

Continuous Awareness Building helps the organization understand the rationale behind the strategy to have **New Features Go Into the New Architecture**. By reinforcing the risks of the legacy system and pointing to the benefits of the new architecture, those involved with product development will agree that creating new features in the old architecture is not a good idea.

Having a clear **Architectural Vision** helps guide the development of features in the new architecture and inspires teams to start using it, speeding up their learning curve. The vision not only provides a high-level overview of the desired end state but also sets forth guiding principles and philosophies that underpin the new architecture and simplifies decisions for teams starting to work on it.

It is essential to **Pave the Road** properly to facilitate the teams' journey into the new architecture and ensure that essential infrastructure, tools, and conventions are in place to support new developments. By having the road effectively paved, teams can confidently develop and deploy new features into the emerging architecture without unnecessary impediments. Training, guidelines, templates, and support will help them become more proficient quickly.

Restrict Changes to the Legacy Application should be used alongside **New Features Go Into the New Architecture** to operate as a protective measure, ensuring that undesired changes do not creep into the legacy system and increase the backlog of the revolution initiative. That pattern provides more details on establishing a process for reviewing which changes should be allowed to the legacy system.

Establishing that **Teams Decide What to Migrate** defines a strategy for taking more control over the prioritization of migration activities yet lets teams steer the efforts according to their point of view, which blends well with **New Features Go Into the New Architecture**

Migrate Critical Subsystems can be combined with **New Features Go Into the New Architecture** to ensure that the revolution initiative addresses risks lurking in parts of the legacy system that are not being actively changed.

Regarding team organization, establishing that **New Features Go Into the New Architecture** goes hand-in-hand with **Assign Architectural Migration to Feature Teams** because the assignment will be straightforward in that case.

The debate about establishing that **New Features Go Into the New Architecture** is especially relevant when the architectural revolution is happening alongside business as usual (as you **Change the Tires of a Moving Car**). However, as discussed in **Freeze**, you may opt to pause the development of new features while migrating to the new architecture. In that case, it is clear that any change request should be implemented in the migrated system after that process is completed. In addition, you should define a review process for changes that can be allowed in the legacy system during the migration. That process would be similar to the one mentioned above and described in more detail in **Restrict Changes to the Legacy Application**.

4. Teams Decide What to Migrate

Credit: Jon Candy On Flickr

The revolution initiative is in motion, with an **Architectural Vision** outlined and some of its foundational components deployed with **Baby Steps**. Teams have been trained in the new technology and are using the templates and elements created to **Pave the Road**. Some squads are already responding to business demands by building new features in the new architecture. Others are migrating components to implement change requests without relying on the legacy system.

You know that it is not enough to rely solely on the arrival of business demands to guide the activities of the revolution initiative. You have **A Plan up Your Sleeve** with milestones and goals, you have chosen **Metrics for Baselining and Comparing** to show the revolution's progress from the customer perspective, and the company may have defined **Company-Wide Architectural Targets** to incentivize the pursuit of those objectives. You also recognize that teams understand in detail the complexity of the legacy system and how it affects their effectiveness.

How do you prioritize migration activities to achieve the architectural revolution's objectives, optimize efficiency, and engage teams?

Autonomy is an influential factor in the motivation of agile teams [Melo et al.]. People feel more committed to a long-term goal when they are free to choose how to achieve it. However, many groups may work in parallel in a revolution initiative, and some may have dependencies between their tasks. As the revolution leader, you want to ensure that everyone involved is working effectively; thus, some coordination is necessary. You also want to provide visibility to the organization. For that, you need information on what has been completed, what is currently being done, and what is planned for execution.

Teams also have deep knowledge about problems with the services they develop and operate. They have been dealing with issues in the legacy system for long enough and know how to avoid these problems in the new architecture. Despite this profound understanding of specific subsystems, teams usually do not have in mind the big picture of the revolution initiative. They may also lack expertise in architectural transformation in general. As a result, teams need guidance on how to measure their progress relative to the revolution's goals, how to approach their migration tasks, and how other teams' activities may affect their outcomes—and vice-versa.

Therefore, let teams decide which components they should migrate and in which order.

Give teams the freedom to define their migration strategy. They should be able to decide the order in which they will migrate components and how to adapt their backlogs to fit those activities. The decision process should involve the whole team, including non-IT representatives such as Product Managers or Product Owners. Teams should also discuss impacts on roadmaps with business leaders so that everyone involved is aware.

Provide guidance on prioritizing component migration based on the goals of the architectural revolution and how each activity will help achieve them. Teams know the major pain points of their domain of responsibility but may not fully comprehend how these issues impact the whole system or the end user. Ensure that teams collect metrics for their components to make more informed decisions on prioritization and assess the results of migration activities. Gather advice from teams that have already succeeded in migrating components and from those involved with the new architecture's infrastructure.

Consolidate all teams' backlogs in a single roadmap to be able to monitor the overall progress of the revolution. Realize that once teams are responsible for their migration strategy, they may have to change and adapt it to their reality. Use portfolio management tactics to closely watch the main milestones, especially those related to critical parts of the legacy system.

When progress on crucial architectural tasks goes sideways, identify what is causing the problem. When you recognize a prioritization issue, resist the urge to interfere with the teams' backlogs. Approach business leaders to reinforce the importance of the revolution and get the teams back on track. If the difficulties persist, consider creating dedicated teams to streamline the migration tasks and reduce their interference with business priorities. When teams struggle to use the new architecture, reinforce training and try to identify improvement opportunities in the new system. Also, consider pairing up a team with more experience in the new architecture to show migration techniques in practice.

✻ ✻ ✻

When **Teams Decide What to Migrate**, they feel more committed and engaged in the architectural revolution. They understand that the goal of transforming the architecture can only be reached if each team does its part. Migrated components are created by those responsible for maintaining and evolving them, dispensing the need for costly hand-offs and knowledge transfers.

The migration process is better aligned with the needs and challenges faced by the teams if they are able to define their priorities. An exclusively centralized decision process would be too slow and costly for a larger group and would not capture the subtleties of each team's reality. A bottom-up prioritization is more effective, because it considers the risks and impacts faced by each team with the legacy application.

Once existing teams are engaged with architectural revolution activities, there is less need for dedicated migration teams, reducing the overall cost of the initiative. It also has the potential to accelerate results, given that more is done in parallel, and it is not necessary to wait for new engineers to be hired and trained to start working on migration tasks.

On the negative side, when **Teams Decide What to Migrate**, there is a potential lack of visibility and consistency across the organization. When there are dependencies between tasks, this situation can cause major issues. To mitigate that risk, the revolution's leaders should have frequent communication with teams and frame the **Architectural Revolution as an Agile Portfolio**. This extra effort can be seen as an overhead when compared to a centralized approach.

Estimates are usually less reliable when **Teams Decide What to Migrate**. Given that teams must continually negotiate with business leaders to accommodate migration tasks in their backlogs, progress may sometimes be erratic. While this can be acceptable in less risky situations, some goals of the revolution need more accurate forecasts. Dedicated teams may be necessary to tackle these cases, as well as to cover parts of the legacy system that are not owned by existing teams.

Regular teams will work part-time on migration tasks and are not specialists in architectural transformation. They need additional support and guidance to make informed decisions about component migration priorities and techniques. They may require advice on how to implement monitoring and define relevant metrics. Thus, letting **Teams Decide What to Migrate** does not mean that they will be handling the architectural revolution solely on their own.

Related Patterns

Having **All in the Same Boat** makes it easier for teams to effectively prioritize architectural migration tasks in their backlogs. That pattern underscores the importance of collective understanding and collaboration across the organization, ensuring that all stakeholders recognize that the revolution initiative is a strategic priority and are aligned in the transformation journey.

Regularly reminding the organization about the importance of the architectural transformation via **Continuous Awareness Building** can encourage teams to actively participate in component migration decisions. If some squads cannot prioritize migration tasks in their backlogs, you may have to reinforce the value of the revolution to those groups.

When teams understand the **Architectural Vision**, they are more effective in prioritizing component migration. They can see which parts of their systems will most benefit from the new style and adjust their migration plan accordingly.

It is crucial to have the **Architectural Roadmap** in sight when **Teams Decide What to Migrate**. The roadmap points to a strategic sequence and priority for migrations designed to alleviate the risks inherent in the legacy system. The ordering suggested in the roadmap can be adapted to some extent depending on the availability of teams to work on their architectural tasks. However, you should keep the most pressing risks in check and frequently assess how the tactical decisions the teams make add up to the big picture of the revolution initiative.

When many teams are involved in migration activities, you have to approach the **Architectural Revolution as an Agile Portfolio** so that you can keep track of relevant tasks and milestones from different teams. You can also track dependencies between teams and resolve them before they cause further delays.

When you **Pave the Road** the migration of components to the new architecture will be smoother, and teams will have more freedom to choose what to migrate. With the proper infrastructure, tools, and guidelines in place, teams will become more confident working in the new architecture and will be more likely to prioritize migration tasks.

Establishing that **New Features Go into the New Architecture** can help **Teams Decide What to Migrate** because it gets them started in working on creating and migrating existing components. They will become more proficient in the new paradigms and will be able to make better decisions on what to migrate. For instance, a good choice is to migrate subsystems that change frequently to make it easier to add new features to those parts of the system.

Sometimes teams might need to **Refactor then Migrate** components they decide are a priority to improve the internal design before engaging in a migration. The decision to refactor before migration should also be a prerogative of the teams. When these patterns intersect, teams are empowered not only to decide what parts of the system are ripe for migration but also to identify where preliminary refinements can expedite and streamline the transition process.

When **Teams Decide What to Migrate**, the primary strategy for team organization in a revolution initiative is to **Assign Architectural Migration to Feature Teams**. The latter pattern proposes that migration tasks be designated to those teams that typically develop features, believing their familiarity with the software's functionality equips them aptly for the migration process. This approach leads to increased knowledge of the resulting components in the new architecture.

Besides that, it is possible to create **Dedicated Migration Teams Within Product Groups** so that some teams can keep developing new features while others are dedicated to architectural work. These migration teams work closely with their peers and can be composed of members from other teams in the same product group. After the migration is complete, the dedicated teams can be dissolved, and reinforce the feature teams, bringing back knowledge of the migration and the new architecture.

5. Migrate Critical Subsystems

Image by Sasin Tipchai from Pixabay

The architectural revolution is at full speed. It is clear to the organization that **New Features Go Into the New Architecture**, so all major change requests are implemented in the new environment. **Teams Decide What to Migrate** and choose what to prioritize in their revolution backlogs in alignment with their business peers.

Meanwhile, the legacy system continues to cause problems for the company, with occasional outages, defects that affect customers, and poor development velocity. The **Architectural Roadmap** indicates which critical subsystems you should migrate that are either currently causing pain or posing potential future risks. However, teams' backlogs may not include some items that should be a priority, suggesting blind spots in the revolution initiative.

How do you ensure that the revolution initiative properly addresses the risks identified in the legacy system?

So far, in the revolution initiative, you have leveraged existing teams and their backlogs to carry out the migration toward the new architecture. This approach can empower teams to decide what and when to migrate, engage them with revolution's goals, and make them responsible for the components created in the new architecture. It also prioritizes the migration of functionality being actively changed so that teams will benefit sooner from a faster development process, shorter deployment cycles, improved monitoring, better availability, etc.

On the other hand, the legacy system may include functionality that was developed long ago and has very stable requirements, so it rarely—if at all—changes. The code implementing these features may be known by only some of the developers currently in the company and may not be actively maintained by any team. Even if a group is assigned to that part of the system, they may be avoiding the migration risk. If these features are critical to the business, moving them away from the legacy system should be a priority of the architectural revolution.

When you consider which risks to mitigate, one option is to address those currently causing disruption in the company's services. This alternative is straightforward and aligns with one important goal of the revolution initiative: to quickly add value by improving areas affected by the legacy system. Focusing on currently unstable subsystems offers immediate relief from pain points. By targeting these areas, teams can directly address the most frequent causes of system downtime, bug reports, and operational inefficiencies. This approach sends a clear message to stakeholders that the migration process is effectively tackling pressing concerns and generating immediate value. Concentrating on these problematic areas, therefore, not only improves system reliability but also builds trust in the revolution initiative.

However, you should also consider other issues in the legacy system that may become existential threats to the company. For instance, a subsystem reaching a hard limit that will cause it to stop working may severely affect the operation. These latent risks, if unaddressed, can materialize in the future—may be unpredictably, causing potentially catastrophic consequences. By proactively migrating subsystems that pose such threats, you can prevent these events and demonstrate that the revolution initiative is concerned with the future of the company.

Therefore, ensure that the teams' backlogs prioritize the migration of subsystems that present critical risks to the organization, balancing current disruptions and future threats.

Evaluate the migration priorities in the **Architectural Roadmap** to decide which are the most critical to tackle first. For each item, consider the disruption it is currently causing to the company from the perspective of the revolution's driving factors, such as service availability, customer satisfaction, operational burden, development productivity, etc. Carefully examine the items that point out threats that may materialize in the future and evaluate their impact. Examples include hard limits that systems may reach, end-of-life of supporting technologies, lack of a proper disaster recovery plan, and security vulnerabilities. These are often overlooked because they may not visibly affect the operation. Still, they can lead to catastrophic events that may be hard or impossible to foresee and may happen without notice.

Take this evaluation and develop a specific migration plan for each prioritized critical subsystem, considering any dependencies, resource requirements, and potential impacts on other parts of the system. You may already have anticipated most of these migrations in the **Plan up Your Sleeve**. Still, in some cases, you may need to ask for some additional budget, especially if you need to assemble a dedicated team.

Define which team will be responsible for migrating a given critical subsystem. You may be able to negotiate with business leaders the reprioritization of an existing team's backlog. Leveraging current squads is usually the ideal solution when enlisting a team already responsible for a domain related to the subsystem in question is possible. When that is not viable, assign a **Specialist Migration Team** for the task and pair it with a team that works close to the corresponding domain so that this domain team will take over the migrated component. Another option is to create a **Dedicated Migration Team Within the Product Group** that is most related to the subsystem domain.

Monitor the migration of each prioritized critical subsystem. Ensure the team responsible has the technical support and the business knowledge necessary to perform the migration. Use **Metrics for Baselining and Comparing** to assess the quality of the new component versus the old version. Given the importance of the process, guarantee that the original functionality is preserved in the new architecture. Unless there is a pressing need to migrate the subsystem as soon as possible, do not rush to the switch to the new version. If necessary, run the updated version in parallel with the old one for some time and compare their outputs to assert the results are the same.

Continually assess whether the subsystems classified as critical in the **Architectural Roadmap** are prioritized in the teams' backlogs and have acceptable forecasts for when they will be migrated. You may have to reach out to business leaders and reinforce the importance of mitigating the risks of the legacy system. If you find that teams are not organically prioritizing the most critical parts of the revolution, identify the root causes, which may include gaps of ownership over some domains, insufficient confidence to tackle challenging migrations, and difficulties in negotiating priorities with business peers.

Consider mitigating the impacts of offending processes when their migration is too complex or cannot be prioritized by the teams in an acceptable time. In some cases, relatively simple measures may pay off in improved stability, customer satisfaction, and better development performance. These tactical actions can relieve the pressure over the revolution initiative and give teams more room to work on a thoughtful migration to the new architecture.

Revisit the **Architectural Roadmap** periodically to check which critical subsystems are prioritized in the revolution's scope. Considering the roadmap is constantly reviewed and updated, the list may change—new risks may appear while others may no longer be relevant.

✻ ✻ ✻

The main goal of **Migrating Critical Subsystems** is to ensure that the revolution initiative addresses the most pressing concerns with the legacy application. Recurrently revisiting the **Architectural Roadmap** and selecting the most critical risks to tackle, you effectively take control of the architectural revolution and guarantee that it moves in the right direction.

As a result, you cover eventual blind spots of the architectural revolution that would be left back if you only establish that **New Features Go Into the New Architecture** and let **Teams Decide What to Migrate**. By systematically identifying processes that offer risk to the business and tackling those that are not addressed in any of the teams' backlogs, you improve the result of the revolution initiative.

A positive side-effect of using this pattern is to increase the organization's understanding of those critical subsystems. Once a core process is identified, mapped, and migrated, not only the risk posed by the legacy system is eliminated, but now a team has fresh knowledge of that process.

On the other side, **Migrating Critical Subsystems** that would not be otherwise addressed in the teams' backlogs means that you have to recruit additional resources to the assignment. Either a team will have to change its backlog to accommodate the effort, a **Specialist Migration Team** will be relocated to perform the migration, or a new team will be created, possibly acting as a **Dedicated Migration Team Within a Product Group**.

Some business leaders may not see value in **Migrating Critical Subsystems** and oppose allocating effort to these tasks. It may be hard for those outside IT to realize the risks involved, especially when the related processes are not causing visible problems. Because of that, it can be more challenging to **Make Progress Tangible** when it comes to migrating those critical subsystems. It may be necessary to reinforce **Continuous Awareness Building** and explain to the other parts of the organization why some activities are crucial to the success of the revolution and the future of the company.

One aspect that must be considered when **Migrating Critical Subsystems** is the inherent risk of changing such vital processes. The revolution initiative would have to tackle this issue anyway, given its goal to reduce the impact of the legacy system. Yet, a word of caution is warranted when using this pattern.

As you move down the list of subsystems not present in any of the teams' backlogs, you will find fewer risky processes. Eventually, you may have left only functionality that is not critical but would have to be migrated so that you can decommission the legacy application. At that moment, toward the end of the revolution initiative, you may have to prioritize the migration of those components or let the legacy system remain active once it no longer offers risk to the organization.

Related Patterns

The **Architectural Roadmap** summarizes major milestones the revolution initiative needs to achieve, along with a chronological guideline. It is the authoritative source from which you should identify the critical subsystems enlisted as motivation for starting the revolution initiative.

Approaching the **Architectural Revolution as an Agile Portfolio** can help you have an overview of the teams' backlogs and identify which critical processes are being left out. Effective portfolio management can also provide up-to-date information on prioritization difficulties, and other obstacles teams face when trying to **Migrate Critical Subsystems**.

You may have to check the **Plan up Your Sleeve** to identify needs for more budget and teams. Especially when you have to **Migrate Critical Subsystems** with no directly assigned squad, you may have to review the plan and seek alternatives.

When teams **Migrate Critical Subsystems** that have not been actively maintained and are in poor technical condition, they may have to **Refactor Then Migrate**. This approach can give squads more confidence in the subsystem's logic before they engage in the migration. It is particularly advantageous if teams are unfamiliar with the requirements.

A **Specialist Migration Team** can play an essential role in **Migrating Critical Subsystems**. Such a team can be responsible for migrating a core process to which no one is assigned. It can also pair with other squads to help them with a critical subsystem, sharing its knowledge in architectural migration.

You may have to reinforce **Continuous Awareness Building** to demonstrate why it is crucial to **Migrate Critical Subsystems**. Highlight the recent events caused by the legacy system and their impacts on the company. Mention the risks with core processes that may not be causing issues but pose threats to the organization's future.

Use **Metrics for Baselining and Comparing** to determine which risks are currently unacceptable to the organization based on the agreed-upon goals of the revolution initiative.

With that in mind, it should be clearer which subsystems to focus on and more straightforward to prioritize the migration.

You should take **Baby Steps** when **Migrating Critical Subsystems**. The best approach to tackling the risk of migration is to do it incrementally. Also, wait until the organization has more expertise in the new architecture and has achieved some **Quick Wins** before approaching complex migrations.

Reluctantly Improve the Legacy Application to mitigate risks in the old architecture and buy time for the revolution to **Migrate Critical Subsystems**. The tweaks in the legacy system can help alleviate some of the impacts on the operation and prepare the ground for the migration of core processes.

Mirroring Features in the New Architecture can make it easier to **Migrate Critical Subsystems**. Teams can run the updated version in parallel with the original. This can be done by using ***Canary Deployment*** [Yoder et al.] to deploy the new version, or just compare the outputs from both versions. When they are confident with the release, they can promote the new version to full production status.

6. Refactor then Migrate

Photo by Alexander Simonsen on Unsplash

Your organization depends on outdated technology and has decided to move to a new architectural style. You have taken **Baby Steps** toward the new architecture, including **Paving the Road**, to make it easier to add new features. Some teams have implemented functionality in the new architecture, achieving **Quick Wins**.

While **Teams Decide What to Migrate**, they may find functionality within the legacy system that would be valuable to be reimplemented or moved to take advantage of the new architecture. However, some pieces are tightly coupled with other slightly related features, making the move troublesome. Although valuable to the organization, other components may be in poor technical condition (with unclear code, high defect rates, low cohesion, minimal test coverage, etc.), and moving them could harm the new architecture.

How do you migrate highly coupled or poorly designed pieces out from the legacy application to the new architecture?

The legacy system may contain very intricate functionality, possibly developed many years ago. These features may be vital to the organization, and migrating them is likely a priority for the revolution initiative. Given this complexity, teams may be uncomfortable reimplementing this functionality from scratch in the new architecture and would rather move them relatively unchanged.

However, these features may be poorly implemented by the organization's current standards, especially considering the quality expected in the new architecture. The code may have low test coverage, high defect rates, and high coupling with legacy infrastructure or other pieces of the old architecture. The team responsible for the feature may only partially understand the logic involved. Replicating this code in the new architecture can compromise quality and create a "broken window" effect, leading other teams to develop substandard components.

Refactoring is an excellent tool not only for improving quality but also for understanding code [Fowler]. Teams facing difficulties migrating complex functionality can use refactoring to make their code more modular, portable, and decoupled from legacy infrastructure and other features. The process has the benefit of creating unit tests, which can be moved to the new architecture and help the teams become more confident with the migration. Moreover, refactoring parts of the legacy application improves its quality, another benefit of the revolution initiative.

On the other hand, teams expect to work primarily on the new architecture and may reject the idea of improving components in the legacy application. Furthermore, some stakeholders may oppose the idea, seeing it as a waste of time and resources, especially given the need to continue developing new features during the revolution initiative.

Teams and business leaders want to conclude the migration as soon as possible to continue creating new features without the burden of the old architecture. Refactoring the legacy system consumes extra time and resources, which may delay the results of the architectural revolution.

Conversely, in some cases, teams may find it simpler to migrate only good-quality code so they will not have to deal with issues in the new architecture. The developers may be familiar enough with the legacy application that organizing some of its parts first can give them extra motivation before migrating.

Therefore, reorganize the parts of the legacy application that relate to the functionality that you want to migrate by refactoring it to a modular design that is easier to replicate in the new architecture.

Help teams identify when they should refactor a functionality before migrating it. This should include an assessment of whether refactoring the functionality—rather than reimplementing it in the new architecture—provides value to the organization. Teams should also assess their level of confidence in the supporting code and its dependencies.

Suggest they evaluate quality attributes of the related subsystem in the legacy application, including test coverage, level of coupling, dependencies between components, and adherence to coding standards, among others [Page-Jones]. These indicators may have been collected in **Metrics for Baselining and Comparing**.

Seams [Feathers] can assist with refactoring the legacy application. A seam is a place to get tests into your legacy system and to help break your code dependencies; they help to alter program behavior without changing the code. There are different seam types: object seams, preprocessing seams, and link seams (or dynamic loading). A common approach is to search for these design seams (***Hairline Cracks***[2]) within the legacy application to extract components so they can be more easily migrated. In other words, look for opportunities to separate functionality and better organize the current system, removing dependencies before you migrate to the new architecture.

It can also be helpful to split the data before migrating the components. Consider whether any data schema that supports the functionality is relatively isolated from other tables and entities in the legacy system database. Find the best way to pull the data apart by finding the weakest links—these indicate ***Hairline Cracks*** that will help you identify domain boundaries in the refactoring.

After finding potential boundaries for the component, start refactoring. Your main goal should be to isolate logic from the legacy infrastructure and other unrelated features. Create plenty of unit

[2] ***Hairline Cracks*** has not yet been written as a pattern, but could be. It is a way to find design seams within your architecture where you can inject tests and possibly split out behavior.

tests focused on validating the domain logic. Well-designed unit tests can be moved with the code to the new architecture and will give the developers more confidence during the migration. Ensure that unit tests are independent of the legacy infrastructure, such as persistence, logging, networking, etc., so they can be ported as-is to the new architecture.

Do not worry about fixing performance issues or improving infrastructure during the refactoring. These should be positive side effects of migrating to the new architecture. Investing in them during the refactoring will lengthen the process and probably will not yield relevant results. Focus on isolating logic from legacy infrastructure and other unrelated features, understanding the domain logic, and creating many unit tests.

✻ ✻ ✻

The core benefit of **Refactor then Migrate** is to avoid the contamination of the new architecture with bad practices from the legacy system. It helps teams get rid of highly coupled components, poor abstractions, and confusing code before they start migrating subsystems with these characteristics. When they actually begin the migration, teams will be less likely to replicate those bad habits because the offending code is now refactored.

Another positive consequence is that teams can become more knowledgeable in potentially complex parts of the legacy system before they start migrating them to the new architecture. The acquired understanding can lead to improved quality of the new components, including higher cohesion, less coupling, and better abstractions.

On the downside, **Refactor then Migrate** can extend the timeline of the migration process. Refactoring, especially when done thoroughly, can be time-consuming, delaying the actual migration and potentially prolonging the period in which teams have to work on the legacy system. There is also the risk of over-investing resources into refining a subsystem that is scheduled to be phased out.

Related Patterns

Any safe refactoring is done by taking **Baby Steps** rather than attempting an extensive redesign. This approach consists of starting with a smaller scope of change, adding many tests along the way, and carefully validating the results before moving on.

When you have appropriate **Metrics for Baselining and Comparing**, it becomes much easier to identify which parts of the legacy application may need refactoring before migration. Teams can use those metrics to decide on the best approach and to justify the need for refactoring.

When **Teams Decide What to Migrate**, they have the freedom to decide how to tackle the migration. They may choose to **Refactor then Migrate** to get enough confidence to transition to the new architecture safely.

Teams that are **Migrating Critical Subsystems** may encounter very complex or poorly designed code that they are not familiar with or that no one has modified in a long time. They may need to **Refactor then Migrate** to get those subsystems in a better shape before adding them to the new architecture.

Given that **New Features Go into the New Architecture**, teams may have trouble migrating the functionality to implement the new requirements. One tool they have is refactoring the corresponding subsystem before migrating it and adding the new feature.

When you **Refactor then Migrate**, you improve modularity, test coverage, and other internal quality attributes, which is a way to **Reluctantly Improve the Legacy System**. Although the goal is to migrate the functionality as soon as the refactoring is done, the legacy system can still benefit from the changes.

Refactor then Migrate provides opportunities for **Quick Wins**, especially if a simple refactoring helps to get some functionality in better shape to migrate it to the new architecture more quickly.

7. Mirror Feature in the New Architecture

Emrah Baysal on Pinterest

More teams have engaged with the architectural revolution, and different migration cases are surfacing. It has been established that **Teams Decide What to Migrate**, and they have chosen to port the features that are significant to the revolution initiative. Some teams chose to **Refactor Then Migrate**, and have achieved positive results even when the original subsystem is poorly implemented.

In some cases, however, a feature that needs to be migrated may have issues not only related to internal properties—such as quality, coupling, cohesion, etc.—but also to external aspects—for instance, a UI (User Interface) or an API (Application Programming Interface). These interfaces directly affect customers and users in general. The team responsible for the feature may want to change the flawed interfaces when migrating to the new architecture.

How do you migrate a feature and change its interface while minimizing the impact on end users?

Decoupling internal migrations from changes affecting end users is generally a good practice. Any technical or infrastructural upgrade is much more flexible if it does not produce any relevant modification in user interfaces—it can be A/B tested, rolled out gradually, and even reverted fully or partially. Ideally, all migrations in an architectural revolution should be decoupled from user interface changes.

Some features in the legacy system may have outdated or poorly designed user interfaces. Migrating those features and keeping their interfaces may cause various issues. For instance, the new architecture may not support older technologies required by those interfaces, or it may be cumbersome to adapt it to do so. There may be plans to update the user experience, and migrating the feature as-is will result in extra effort. Moreover, teams may feel demotivated to implement an inadequate user interface alongside modernized components in the new architecture.

Therefore, create an improved feature in the new architecture and keep the original available so that users can compare both versions and switch to the new feature when convenient.

Help teams identify which features will benefit from the mirroring technique. The ideal scenario for the approach is as follows. The feature has an interface (UI or API) you do not want to preserve when migrating to the new architecture. That is the case when the interface is incompatible with the new architecture’s technology or domain model, and it would demand significant effort to adapt it. Besides, you do not want to force a new interface on the users because it would require them to learn a different UI or integrate with an incompatible API. As a result, the strategy is to replicate the feature in the new architecture, leave both versions available for a while (the old interface in the legacy system and the new interface in the replicated feature), and invite users to try the new one.

If the old interface is incompatible with the new architecture, but changing to the desired new interface will not cause major issues to the users, the team can migrate the feature and change its interface in the process. On the other hand, if adapting the old interface to the new architecture is viable, it is recommended to migrate the functionality as-is and only consider changing the interface later, when the feature is already stable in the new architecture.

Once it has been decided that a particular feature will be mirrored, develop it in the new architecture. From the user's perspective, the new feature should be equivalent to the old one. In other words, they should perform the same user goal, even if they are entirely different in interface and implementation.

Make both versions compatible regarding their internal data so that users can switch back and forth between them without losing any information. Their data models may be very different, but with some form of replication between them, both versions can coexist. Before rolling out the new version, develop a switching functionality enabling users to choose which one they prefer.

Deploy the mirrored feature and define which users can test it, rolling it out progressively in **Baby Steps**. For the selected users, let them choose the new version and switch back to the old one. Offer help to the potential users, such as a guided tour (for a UI) or documentation (for an API). Include some form to get feedback and use the responses to adapt and improve the new feature.

Plan to decommission the old feature. Consider leaving it active for the time necessary to get feedback and improve the new version. Also, assess how long users need to adapt. For a UI, this can be a few weeks to a few months, but more time may be necessary for an API. Communicate periodically with users when the feature is close to being removed. Even with frequent warnings, some users may lag behind and not switch to the new version. You may have to consider turning the feature off, even with some active users, as a last resort.

✻ ✻ ✻

When teams **Mirror a Feature in the New Architecture**, they are free to create an improved version that may be very different from the original one. The improvements may reflect updated domain knowledge, better customer insights, and user interface upgrades. This freedom boosts the team’s motivation because they are not forced to migrate or reimplement a flawed user interface just for compatibility with the old version. Teams also have the opportunity to gather valuable feedback from users to improve the new version of the feature.

Users also benefit from having a smoother transition to enhanced functionality. They can test and compare both versions for some time, gradually adapting to the new version. This ability is essential when the feature is accessible via an API—having both versions available for use in parallel leads to better testing and a safer upgrade.

Drawbacks from **Mirror Feature in the New Architecture** include the additional resources and effort required to maintain both old and new versions of the feature during the transition period. Data compatibility can also be challenging, especially for functionalities that include creating, reading, updating, and deleting (CRUD) data. Decommissioning the old version can take a long time and be traumatic to some customers, particularly regarding APIs. This effect has the potential to delay the overall architectural transformation process.

Related Patterns

The organization will make more informed decisions about whether or not to **Mirror a Feature in the New Architecture** when **Teams Decide What to Migrate**. Empowering teams to decide what and when to migrate can help identify features that no longer align with current organizational goals or require mirroring in the new architecture.

A clear **Architectural Vision** helps teams understand what technologies and domain models are compatible with the new architecture and have better decisions on when to **Mirror a Feature in the New Architecture**.

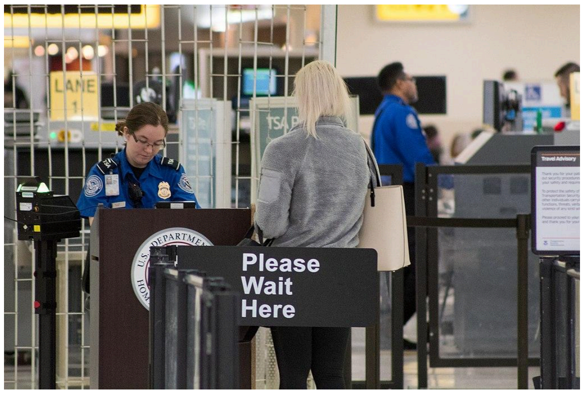
After establishing that **New Features Go Into the New Architecture**, teams should not try to evolve features in the old architecture and instead may decide to **Mirror a Feature in the New Architecture**.

Mirroring Features in the New Architecture can help you **Restrict Changes to the Legacy Application** by locking the feature in the legacy system. You should only allow changes to that version when strictly necessary (bug fixes, security patches, etc.) and you should ensure that the version in the new architecture remains compatible after the changes.

It is often helpful to **Mirror Features in the New Architecture** when you have to **Migrate Critical Subsystems**. The parallel versions make it easier to test the migrated subsystem and compare the results, giving you more confidence on switching to the new version.

You can use ***Canary Deployment*** [Yoder et al.] with the new version so that it is released gradually to users that will be able to test it, and then choose it as their default option. If there are no major issues and users consistently opt for the new version, it can be rolled out to the remaining users.

8. Restrict Changes to the Legacy Application



Michael Ball, CC0, via Wikimedia Commons

The organization has decided to evolve to a new architecture. You have **Paved the Road** to simplify its adoption, and teams have started taking some **Baby Steps** toward the new architecture.

You have mandated that **New Features Go Into the New Architecture** to avoid creating more complexity in the legacy application and to stop increasing the backlog of the revolution initiative. You recognize that teams may have to implement urgent changes in the legacy system, and you need a process to ensure that they only do so when authorized.

How do you prevent teams from adding new features in the old architecture, yet still have a way to decide which critical changes should be implemented in the legacy application?

During the potentially long-running process of evolving to a new architectural style, it is natural that developers and especially product managers feel inclined to add functionality to the current system. Doing so is typically faster and less expensive than providing the same functionality in the new architecture. Teams may be incentivized by business peers to take the shortest path and finish their tasks sooner by implementing in the legacy application.

On the other hand, if teams are free to keep adding and changing features in the legacy system, the revolution initiative may never reach its goals. If there are no design standards or policies set forth to require new or modified functionality to be created in the new architectural style, the current system may see occasional growth despite efforts to reduce its scope.

Many teams have been working on the current architecture for a long time. Some of them may not fully engage in development for the new architecture, perhaps because they are too focused on feature creation and are not acquainted with the new technologies and tools. These groups are more prone to keep adding code to the legacy system and not reap the benefits of the new architecture. Others may realize that the new architecture provides new technologies they could benefit from, but are unsure about how to use it, especially because this might involve integration with some parts of the legacy system.

In spite of that, some developers and product managers feel the urge to take advantage of the new architecture. They see potential technical and business benefits and realize that, after overcoming the challenges of understanding the new architecture and migrating to it, they will become more productive, they will be able to use technologies not available in the current system, and the quality of their service will improve.

You have a mandate stating that **New Features Go Into the New Architecture**, which directs teams to stop using the legacy system for creating or changing functionality. This step is necessary to ensure that the organization switches to the new architecture as the default place to create value for its customers. Although this directive may cause delays to some planned activities, you expect that most initiatives will be able to accommodate the impacts, especially considering future benefits, such as the risks avoided in the legacy application.

However, you recognize that some small and urgent changes may have to be made directly in the legacy system to avoid organizational risks or customer impact. Some examples are defects that are affecting customers, security fixes, and simple features that may bring competitive advantage. Implementing those changes in the new architecture may take a disproportionate time and expose the company to more risks.

Therefore, establish a directive that proposed changes to the legacy system must be approved before they are implemented.

Include a mechanism in this directive that exposes and highlights the changes proposed to the legacy system with evidence on why each change should be made. Ensure this mechanism will have a process for reviewing the proposed changes with the overall organization in mind, allowing only the most critical (necessary) changes to be done in the old architecture. The review process can use a *governance committee*. This committee includes representatives from both IT and other areas of the organization, such as Products, Business, Sales, and Marketing. They need to be aware of the risks of the legacy system and agree on the importance of the architectural revolution. The committee's goal is to review the proposed changes, discuss impacts versus opportunities for the whole organization, and decide which can be done in the old architecture.

This governance committee can be a *Change Advisory Board* (CAB), as described by the ITIL Framework [Aguter]. Although CABs are usually associated with waterfall and highly staged processes—and thus incompatible with agile methods and DevOps [Forsgren et al.]—they can work in favor of the revolution initiative. By forcing a discussion about prioritizing changes to the legacy system at a more strategic level, the CAB ensures that any proposed change is scrutinized for its genuine urgency and value. This vetting process inherently challenges teams to justify any proposed modification to the legacy system, compelling them to consider the broader implications for the architectural revolution initiative. These discussions involve higher ranks of the organization within the CAB, so they are evaluated more strategically, ensuring that only the most critical changes are approved.

Figure 4 depicts the mechanics of implementing a new requirement when the organization uses a CAB to **Restrict Changes to the Legacy Application**. The team responsible can always implement a new feature using the new architecture. However, if they believe that the new requirement needs to be implemented in the legacy system, they must submit a proposal to the CAB. The CAB then decides whether to reject the change and demand that the team implement it in the new architecture, approve the development in the old architecture, or postpone the decision by requesting more details or alternatives from the change proponents.

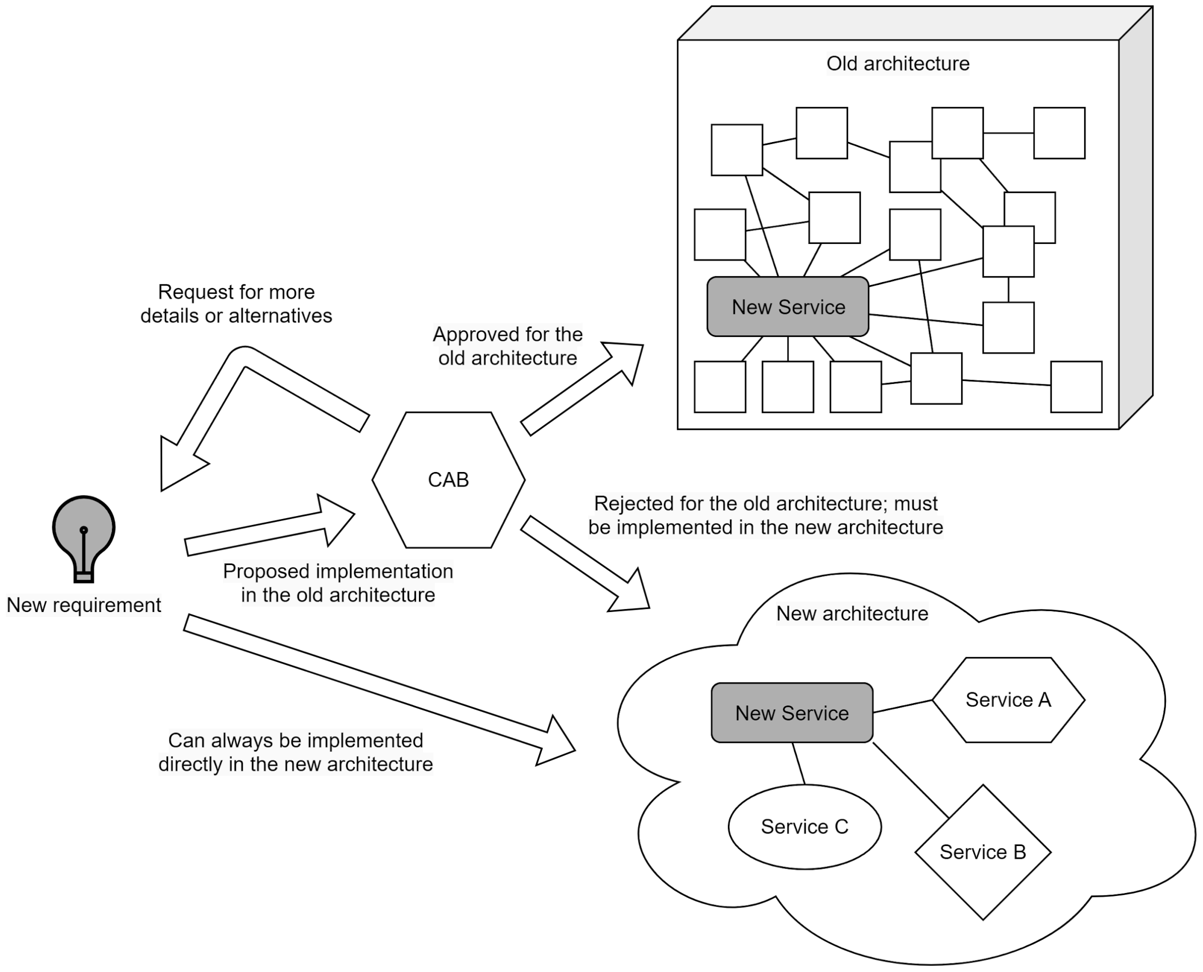

Old architecture
New Service
Request for more details or alternatives
Approved for the old architecture
CAB
Rejected for the old architecture; must be implemented in the new architecture
New requirement
Proposed implementation in the old architecture
New architecture
New Service
Service A
Can always be implemented directly in the new architecture
Service C
Service B

Figure 4: Mechanism for Restricting Changes to the Legacy Application

Ensure that the members of the CAB are carefully selected to make it as effective as possible. Consider the following roles: the architectural revolution leader, an IT executive (ideally the CTO), and a lead business or product development executive, at least when there is an important discussion related to their respective department. The people proposing each change to the CAB should be those involved with the initiative requiring the modification. Ideally, there should be one person from a business or product-oriented role and one with a technical background presenting the change. The former can explain the motivation, while the latter should clarify what modifications will be done to the legacy system. The CAB then evaluates the importance and urgency of the change and contrasts it to the risk it involves and the complexity it will add to the legacy system. After discussing the matter, the CAB should decide whether to approve the change, reject it, or postpone the decision, asking for more details or alternatives. If there is no consensus within the board, the discussion may have to scale up to the top level of the organization (eventually to the CEO).

CAB meetings are usually rich opportunities for understanding what type or requirements usually impact the legacy application. Sometimes, the change requests are simple, superficial, not related to requirements, like a configuration, or related to a feature that many teams depend on (one example from a case study is a menu in a backoffice screen that had to be updated for every new feature developed in the organization). In that case, a relatively simple change in the legacy application could help teams to become more independent. In other situations, the dependencies are related to core processes of the organization which may affect many teams. Identifying those dependencies may help identify opportunities for prioritization in the **Architectural Roadmap.**

One organization evolved their governance process over time, introducing more restrictive rules for changes to the legacy application:

- *First phase*: At the outset, the organization established that teams should use only the new architecture for implementing requirements but did not enforce that recommendation. Therefore, any team could still create features in the legacy application and roll them out without restrictions.
- *Second phase*: After some time, the organization created a CAB and mandated that teams could only implement new features in the legacy application after approval by the CAB. Only after the changes were approved could the requesting team develop and deploy the modifications.
- *Third phase*: Later on, teams still had to submit all legacy system change proposals to the CAB and could start implementing them after approval. However, after the development was ready, the **Team Owning the Legacy Application** had to review and approve the changes before the team responsible could roll them out to production.
- *Fourth phase*: In the last step, teams submitted legacy system changes to the CAB, and if they were approved, then the **Team Owning the Legacy Application** would implement and deploy the modifications.

Besides that, this organization defined some change requests that did not require approval by the CAB (pre-approved requests): bug fixes, security patches, and changes aimed at migrating features to the new architecture. The CAB would meet weekly, but it was also possible to submit urgent requests anytime—this is where leaders evaluate the request as soon as possible so that the requesting team does not have to wait until the next weekly meeting.

✻ ✻ ✻

The main benefit of **Restricting Changes to the Legacy Application** is to protect the architectural revolution by ensuring that the situation will not become worse in the current system. If new functionality were allowed freely in the legacy application, the revolution initiative would have to chase a moving target because its backlog would be increasingly large.

An additional reward for the organization is an increased consciousness about the importance of sound architecture and the confirmation of the revolution initiative's strategic relevance. The discussions held by the governance committee, which may involve top business leaders, help bring the theme of systems architecture to the high ranks of the organization. This new understanding may lead the company to a new approach to software development.

Another advantage when you **Restrict Changes to the Legacy Application** is that the organization can expedite the migration to the new architecture, thus reaping the benefits throughout the organization sooner.

However, one possible downside of **Restricting Changes to the Legacy Application** is that it could delay initiatives estimated under the assumption that they would be implemented in the legacy system. Initially, it could take longer to implement features in the new architecture because you cannot quickly add behavior the same way you did in the old system by using simple techniques like copy-and-paste. This situation may cause a backlash from business leaders and product managers, who will either ask for their demands to be developed in the legacy application or criticize the revolution initiative altogether.

Finally, there is time and effort in setting up a governance committee, which can slow down development efforts that need to go through the new approval process. Developers and product managers may criticize the approach as bureaucratic and anti-agile.

Related Patterns

When you decide to **Freeze** new feature development during an architectural revolution, you should establish a way to **Restrict Changes to the Legacy Application**. Otherwise, it will be difficult to guarantee that the **Freeze** is effective. Without restrictions, well-intended teams might inadvertently introduce changes, believing them to be valid exceptions or urgent requirements. This behavior can lead to inconsistencies, potential instabilities, and deviations from the main objective of the revolution. Having both these patterns in place ensures that **Freeze** is not just a mandate but is reinforced by practical and controlled processes.

Restricting Changes to the Legacy Application is also necessary when you **Change the Tires of a Moving Car** and have new features developed alongside the architectural revolution. In this dynamic environment, without proper guardrails, it is trivial for teams to cross the line and implement new functionality in the old architecture, leading to more potential risks and increasing the revolution's backlog. By strictly delineating and governing the changes permitted on the legacy system, you ensure that every change aligns with the revolution initiative's goals—while the car is still moving.

You can take **Baby Steps** while **Restricting Changes to the Legacy Application**. You can start by adopting a lighter reviewing process and then evolve to stronger restrictions to the legacy application.

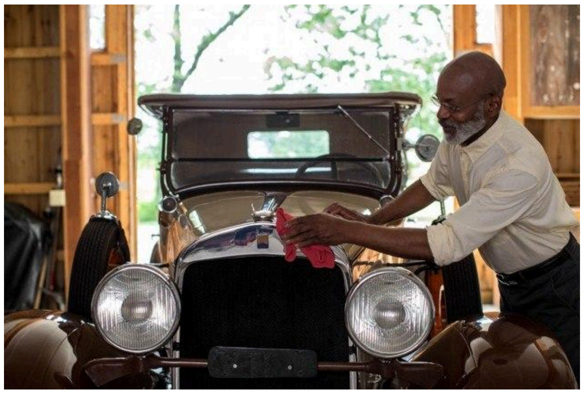
While implementing a new requirement in the legacy application, teams will sometimes notice opportunities where they can **Refactor then Migrate** the legacy system, thus allowing functionality to be more easily migrated to the new architecture. These refactorings should be considered changes that remove complexity from the legacy system, and would not have to go through the governance committee for approval.

While you are **Restricting Changes to the Legacy Application**, it is still important to **Reluctantly Improve the Legacy Application**. It is a mistake to ignore the old architecture especially because it will be around for a while and it still provides value to the organization. The changes you make in the legacy application to improve it may not need approval, provided that you ensure that they only focus on non-functional aspects and do not add any new capabilities that will have to be reimplemented in the new architecture.

When you **Restrict Changes to the Legacy Application**, it becomes clear that only urgent changes can be made to the old architecture, and teams should work primarily in the new environment, which may lead them to decide to **Mirror a Feature in the New Architecture**. This approach leads them to work more freely in the mirrored feature and keep the original one static in the legacy system.

It is essential to let **Teams Decide What to Migrate** when you **Restrict Changes to the Legacy Application**. When the door to the old system is no longer freely available, teams may have to adapt and choose to migrate a frequently changing subsystem to keep evolving its features.

9. Reluctantly Improve the Legacy Application



Credit: Nationwide blog (https://blog.nationwide.com/hot-rod-maintenance/)

You have had some success taking **Baby Steps** toward a new architecture, realizing the **Architectural Vision**. Teams have achieved **Quick Wins**, and some are readily accepting the new architecture. There is growing development within the new architecture, especially by having **New Features Go Into the New Architecture** and the mandate to **Restrict Changes to the Legacy Application**.

However, the legacy system still provides value to the organization, and teams need the old architecture to be in good shape to migrate functionality. Besides, the old architecture presents risks that can cause severe impacts if not properly mitigated.

How do you keep the legacy system serving its purposes well enough so it does not hinder the architectural revolution?

The organization has decided to migrate the functionality from the legacy application to the new architecture. An architectural revolution is on course, and teams are focused on developing features in the new model. Because teams and resources are limited, you may not feel that it makes sense to invest in improving the legacy system.

However, neglecting the old architecture is a mistake, especially if core processes still depend on it and migration will not be completed soon. Customers will still be susceptible to slow response times, outages, and other known issues. Experienced engineers will be involved in incident resolution and will not be able to work on the revolution while firefighting. The more problems you have with the legacy system, the more pressure business leaders and executives will put on you to hurry with the architectural revolution—or worse, they might get frustrated and want to cancel the initiative.

Furthermore, the legacy application will be going through frequent changes during the revolution because existing subsystems may have to be integrated with new components, features may have to be refactored before migration, etc. So, if it is painful to modify the legacy system's code, test the changes, or deploy the results, this situation will impact the revolution initiative.

Therefore, mitigate the major pain points of the legacy application to reduce risk, improve productivity and relieve pressure on the revolution initiative.

Identify the most critical issues in the legacy application based on their potential impact on the business, end users, and teams. Prioritize those that can be mitigated with relatively low effort. The goal is to have **Quick Wins** on this front and keep the old architecture under control until the most critical features are migrated. Do not try to solve the root cause of the problems—this is being addressed by the revolution initiative itself.

Establish a dedicated team to undertake the improvements on the legacy system. At a minimum, allocate part of an existing team's time to address the identified mitigations. Find developers that understand the old architecture and are willing to dedicate their time to working on it. Motivate these developers by highlighting the importance of their mission—it is not a throwaway work but an essential activity to protect the revolution initiative.[3]

Focus on improving system capabilities such as monitoring, troubleshooting, modularity, automated testing, and deployment. These will usually pay off more than focusing on specific issues because they will influence the system as a whole and all teams working on it.

Communicate the importance of maintaining the legacy application to stakeholders and other teams involved in the architectural revolution. It is usually hard to recognize the value of preventive work, and you need to protect the dedicated team from being relocated to other priorities.

✻ ✻ ✻

When you **Reluctantly Improve the Legacy Application**, you reduce the risk of critical issues and disruptions during the architectural revolution. As a consequence, you have an improved user experience and satisfaction for those still depending on features running on the old architecture.

With fewer incidents, stakeholders may feel increased confidence in the architectural revolution initiative. A sense of stability can provide extra motivation to everyone involved. Teams working on migration can focus on their backlogs and worry less about firefighting production problems. Although some of the improvements may be temporary or superficial, they help protect the initiative from interruptions and potentially dangerous resource discussions.

On the other hand, relocating experienced developers to work on the legacy system competes with the migration to the new architecture, which could slow down the revolution's progress. This situation can be questioned by business leaders, and you may have to justify the importance of **Reluctantly Improving the Legacy Application**.

Moreover, if you expect your architectural revolution to be relatively fast, maybe there is no point in worrying about improving the legacy system during that time. Assign all developers available to migration tasks and get the legacy system decommissioned as soon as possible.

Related Patterns

By using **Metrics for Baselining and Comparing** to evaluate the performance and quality of the legacy application, you can determine the areas where improvement is most needed and

[3] This can have some dangerous implications. Look at the consequences for addressing this.

prioritize them accordingly. These metrics will provide information for reporting progress on the improvements.

The goal of **Reluctantly Improving the Legacy Application** is to care for its non-functional attributes, such as stability, availability, maintainability, and deployability. It is essential to **Restrict Changes to the Legacy Application** and define clear rules and boundaries for changes that can be made to the old architecture to avoid contradicting the goals of the architectural revolution.

When teams **Refactor Then Migrate**, they also improve internal aspects of the old architecture, such as maintainability and modularity, at least in specific subsystems. These teams are also helping to improve the legacy application.

As you **Reluctantly Improve the Legacy Application**, look for some **Quick Wins** to address some risks and pains as soon as possible. This will motivate the teams involved and help justify the investment.

A **Team Owning the Legacy Application** is the ideal option to take responsibility for the improvements. With a dedicated team, the legacy system will have constant attention and focus. Even if incidents occur, the team can be the primary responder, shielding other teams working on feature development and migration tasks.

You should **Make Progress Tangible** to highlight the improvements on the legacy system. It is likely that most of the people in the organization will not intuitively understand the benefits of supporting the old architecture. Showing the progress in clear and objective terms can help justify the effort put into it.

10. Putting It All Together

Figure 5 depicts the relationships involving the patterns presented in this paper. The patterns in this subgroup ("Tactical Prioritization") are shown inside the dotted line. Patterns from the related subgroup "Strategic Prioritization" are in black, outside the dotted line. Other patterns from our language (described in other works) are shown in gray (patterns referenced multiple times are marked with an asterisk). Patterns from related works are cited in gray and italics, with the source referenced in square brackets and brief descriptions (patlets) provided in **Appendix A**. The relationships between patterns in the language are explained by labels close to the lines and should be read according to the direction of the arrow.

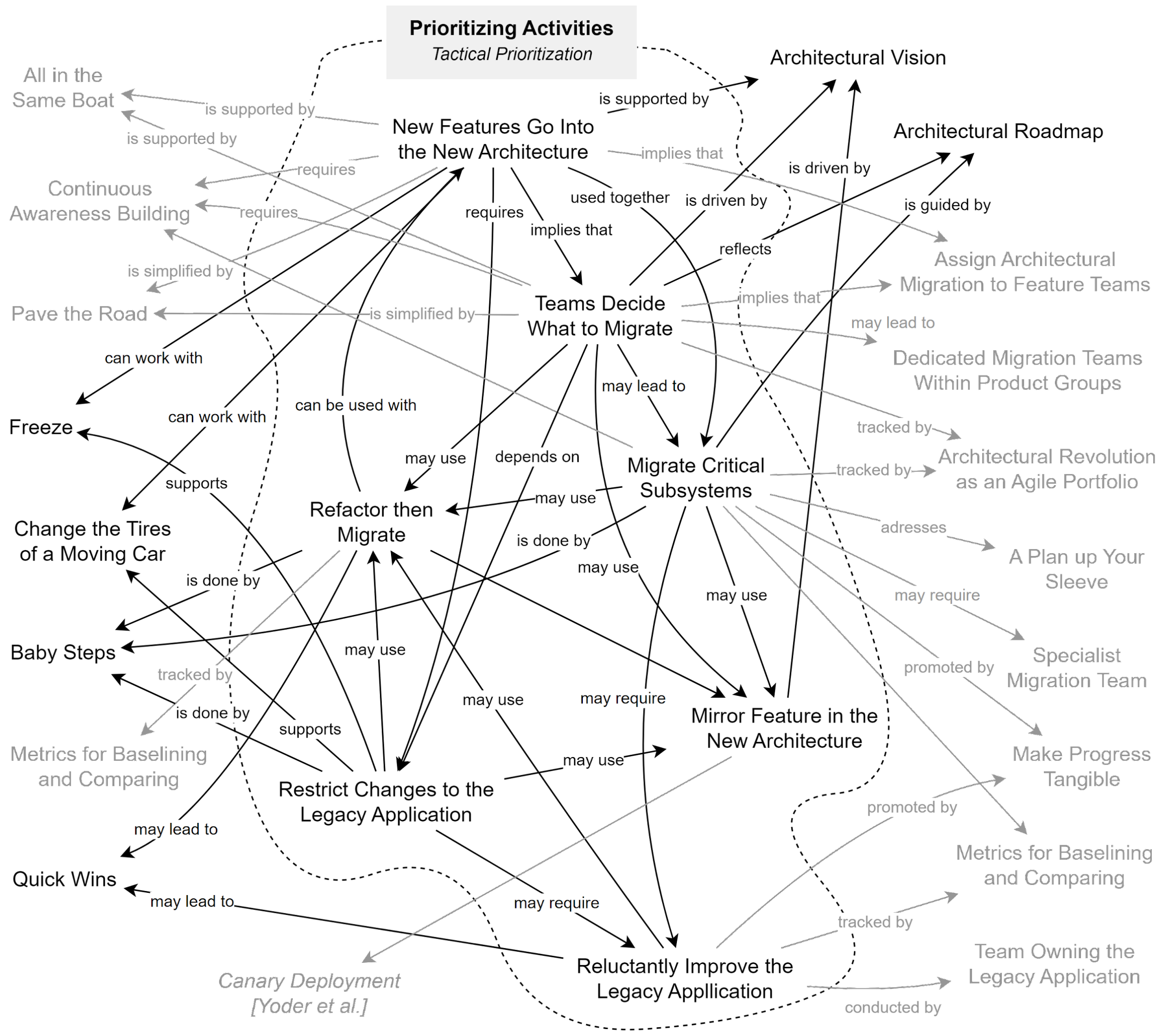

Prioritizing Activities
Tactical Prioritization
All in the Same Boat
Continuous Awareness Building
Pave the Road
Freeze
Change the Tires of a Moving Car
Baby Steps
Metrics for Baselining and Comparing
Quick Wins
New Features Go Into the New Architecture
Teams Decide What to Migrate
Refactor then Migrate
Migrate Critical Subsystems
Restrict Changes to the Legacy Application
Mirror Feature in the New Architecture
Reluctantly Improve the Legacy Appllication
Canary Deployment [Yoder et al.]
Architectural Vision
Architectural Roadmap
Assign Architectural Migration to Feature Teams
Dedicated Migration Teams Within Product Groups
Architectural Revolution as an Agile Portfolio
A Plan up Your Sleeve
Specialist Migration Team
Make Progress Tangible
Metrics for Baselining and Comparing
Team Owning the Legacy Application
is supported by
requires
is simplified by
can work with
can be used with
supports
is done by
tracked by
may lead to
may use
depends on
used together
implies that
is driven by
reflects
is guided by
tracked by
adresses
may require
promoted by
conducted by

Figure 5: Patterns for prioritizing activities (tactical prioritization)

To effectively prioritize activities during an architectural revolution, first you must ensure that **New Features Go Into the New Architecture** to prevent the legacy system from growing in complexity during the revolution. Enforce that rule even if it forces you to migrate a large subsystem to add a smaller feature—you can handle exceptions to the rule if necessary.

Teams know best what affects their daily work, so it is valuable to have them drive the prioritization of their architectural tasks by letting **Teams Decide What to Migrate**. When teams are responsible for developing components in the new architecture they become more familiar with the technology and get a sense of ownership over the new pieces of software.

On the other hand, you may have to deliberately prioritize the mitigation of risks in the old architecture and **Migrate Critical Subsystems**. That may happen when some parts of the legacy systems are not being actively changed and have not been selected by any team for migration. If those pieces are causing issues or may pose future risks, you should find a way to prioritize their migration to the new architecture—either assigning it to an existing team or creating a new team to handle it.

Sometimes the legacy application code is so poor that it pays off to **Refactor then Migrate** a subsystem. Although this approach is not always applicable, it may help teams that are not familiar with the rules and logic of a particular subsystem. By refactoring they get to know better that part of the application and take smaller steps, improving the design before migrating to the new architecture. As a result, the code may be ported with less changes to the new system.

When migrating a certain feature will change its interface and potentially affect customers, it may be necessary to **Mirror Feature in the New Architecture** and keep both versions working in parallel for some time. Customers will be able to use both versions in parallel and decide when to switch to the new version. You can also benefit from the ability to compare results and to revert to the old version if there is anything wrong with the new one.

To ensure that the teams are not inadvertently creating or changing features in the old architecture, **Restrict Changes to the Legacy Application** by creating gatekeepers to avoid undesired modifications. Establish clear criteria around which changes should be allowed to the legacy system and consider forming a committee to decide on corner cases.

Finally, **Reluctantly Improve the Legacy Application** to avoid increasing risks and keep productivity reasonable when teams have to modify the legacy system. Focus on mitigating the most critical risks without investing significant effort. Find ways to avoid incidents and other interruptions that reduce the teams’ focus on the revolution.

11. Conclusion

In the evolving landscape of software development, a system architecture may gradually (or abruptly) become inadequate given new market needs and organizational transformations. When incremental adjustments prove insufficient to realign the software architecture with these new realities, a committed and comprehensive restructuring is needed. This process, which we term as an “architectural revolution” in software systems, calls for a deliberate and methodical approach. This paper adds to a series of publications that set forth a pattern language designed to elucidate and facilitate this intricate process of change.

We previously described patterns to help leaders deal with large changes to software architecture. Our first paper described the topics of “Creating Awareness” and “Preparing and Measuring” [Neubert, Yoder 2022]. Our second paper inspected strategic prioritization decisions that should be taken when beginning the architectural revolution [Neubert, Yoder 2023]. The patterns described in this article explore tactical alternatives to prioritization, focusing on selecting activities from the revolution backlog, assigning them to teams, and keeping teams focused on the most valuable tasks.

Future work will present in detail other patterns for organizing teams for more effectiveness in a software architecture revolution (these are the last patterns in this language and are outlined as patlets in Appendix A).

12. Acknowledgements

We would like to thank our shepherd Richard P. Gabriel for his valuable insights, comments, and feedback during the PLoP 2023 shepherding process. Richard spent many hours helping us with this paper and the previous related papers. His shepherding included many ideas to better outline and describe the patterns in our paper. He also encouraged us to experiment with various ideas to better outline and describe the patterns for our paper. This work was partially done by one of the authors (Joe) while collaborating with USP-IME.

We would also like to thank our 2023 PLoP Writers Workshop Group: Indu Alagarsamy, Andrew P. Black, Douglas C. Schmidt, Mary Shaw, and Kurihara Wataru for their valuable comments and suggestions during the writers workshop of our paper.

13. References

[Aguter] Agutter, C. (2020). ITIL Foundation Essentials ITIL 4 Edition-The Ultimate Revision Guide. IT Governance Publishing Ltd.

[Alexander et al.] Alexander, C., Ishikawa, S., & Silverstein, M. (1977). A Pattern Language. Oxford University Press.

[Bass et al.] Bass, L., Clements, P., & Kazman, R. (2021, August). Software architecture in practice, fourth edition. Addison-Wesley Professional.

[Cohn] Cohn, M. (2005). Agile estimating and planning. Pearson Education.

[Feathers] Feathers, M (2005). Working Effectively with Legacy Code. Addison Wesley.

[Forsgren et al.] Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations.

[Fowler] Fowler, M. (2018). Refactoring. Addison-Wesley Professional.

[Kerzner] Kerzner, H. (2018). Project management best practices: Achieving global excellence. John Wiley & Sons.

[Melo et al.] Melo, C. D. O., Santana, C., & Kon, F. (2012, September). Developers Motivation in Agile Teams. In 2012 38th Euromicro Conference on Software Engineering and Advanced Applications (pp. 376-383). IEEE.

[Neubert, Yoder 2022] Neubert, M., & Yoder, J.W. (2022, October). Leading a Software Architecture Revolution - Part 1: Creating Awareness, Preparing and Measuring. In HILLSIDE Proc. of 29th Pattern Languages. of Programs Conference (PLoP 2022).

[Neubert, Yoder 2023] Neubert, M., & Yoder, J.W. (2023, July). Leading a Software Architecture Revolution - Part 2a: Strategic Prioritization. In Proceedings of the 28th European Conference on Pattern Languages of Programs (pp. 1-20).

[Page-Jones] Page-Jones, M. (1995). What every programmer should know about object-oriented design. dimensions, 227(29), 252-343.

[Seacord et al.] Seacord, R. C., Plakosh, D., & Lewis, G. A. (2003). Modernizing legacy systems: software technologies, engineering processes, and business practices. Addison-Wesley Professional.

[Yoder et al.] Yoder, J. W ., Aguiar, A., Merson, P., Washizaki, H., “Deployment Patterns for Confidence,” 8th Asian Conference on Pattern Language of Programs (AsianPLoP), Tokyo, Japan, 2019.

Appendix A – Software Architecture Revolution Patlets

The following is a brief discussion of the entire collection of patterns using patlets in the tables below. A patlet briefly outlines the gist of a pattern, usually in one or two sentences.

Creating Awareness

You cannot assume that everyone in the organization understands the issues with the current architecture, so you must level that understanding by showing practical effects in the product development cycle, the risks the organization is incurring, and what can be done to improve the situation.

Patlet Name	Description
Awakening	Detect and become aware of the problems with the architecture and acknowledge it needs to be transformed with a dedicated effort.
All in the Same Boat	Bring stakeholders together to participate in the daily routine of IT teams so that they understand technical challenges and share responsibility in decisions. Also, have IT leaders participate in strategic discussions, learn more about the company, and contribute with ideas.
Continuous Awareness Building	Discuss the issues with the existing architecture frequently and at various levels of the organization, highlighting the risks lurking ahead and pointing to the proposed solution.
A Plan up Your Sleeve	From early in the planning stage of the architectural revolution, have a plan ready for pitching the initiative, including a vision of the new architecture and a rough roadmap.
Scare the S* out of Them**	When an opportunity arises to have the undivided attention of management to talk about the issues of the current architecture, be as assertive as possible to show them the risks it represents for the organization, both in the short and the long term.

Preparing and Measuring

As you seek approval to start a revolution, start preparing the infrastructure for teams to work on. It is also the time to define which measurements will track your improvements and create processes to gather them. Ideally, some of these will become targets for the company. Manage the initiative by treating it as an agile portfolio and reporting its results in a transparent way.

Patlet Name	Description
Pave the Road	Make it easier to develop features in the new architecture by training teams, hiring dedicated people, and providing the fundamental environment for building and deploying applications.
Metrics for Baselining and Comparing	Choose a set of metrics for baselining the legacy application and comparing the new architecture against it. Use those metrics to evaluate the progress of the revolution initiative and report them to the organization.
Company-Wide Architectural Targets	Create company-wide targets for the architectural revolution. Use these targets to signal that the revolution initiative is an important strategic goal and to help teams prioritize architectural work.
Make Progress Tangible	Adopt an accessible style for communicating the progress of the revolution initiative. Translate technical concepts to visual or physical representations that anyone in the company can understand.
Architectural Revolution as an Agile Portfolio	Treat the architectural revolution as an agile portfolio. Assign a portfolio manager to the initiative and give them access to all teams involved.

Prioritizing Activities

You want to deliver value as soon as teams start working on the revolution initiative. Prioritizing which activities to engage first according to the reality of your organization is crucial for ensuring that internal stakeholders will quickly see results from the new architecture. Prioritizing activities can be broken down into "Strategic Prioritization" and "Tactical Prioritization" as outlined below.

Strategic Prioritization Patterns

Patlet Name	Description
Architectural Vision	Define the new architecture in a clear and inspiring way so that teams will know what to expect and be motivated to migrate.
Architectural Roadmap	Create a roadmap for the new architecture which includes when various architectural features should be addressed. This roadmap prioritizes the deployment of the architecture and when to migrate critical pieces.
Baby Steps	Take small steps toward the new architecture. Provide just enough infrastructure to support the simplest cases and validate the vision.
Quick Wins	Migrate simpler, less risky, and less coupled subsystems before you engage in more complex activities so teams can learn about the new architecture and the migration process.
Freeze	Stop developing features while teams migrate to the new architecture to achieve faster results in the revolution.
Change the Tires of a Moving Car	Reconcile architectural migration with feature development to keep the business evolving during the course of the revolution.

Tactical Prioritization Patterns

Patlet Name	Description
New Features Go Into the New Architecture	Avoid increasing the backlog of the architectural revolution by establishing that teams should only develop new features or change existing ones in the new architecture.
Mirror Feature in the New Architecture	Create an improved version of a feature in the new architecture while keeping the original available so that users can compare both versions and switch to the new feature when convenient.
Teams Decide What to Migrate	Let the teams engaged in a particular subsystem decide which components they should migrate and in which order.
Refactor then Migrate	Refactor the legacy application to more easily migrate a subsystem to the new architecture.
Migrate Critical Subsystems	Migrate subsystems that are causing disruption or increasing risks for the legacy application, even if they are not being actively changed.
Restrict Changes to the Legacy Application	Allow some changes to the legacy application while defining objective criteria that must be met to justify each request and assigning a review board to discuss the candidates.
Reluctantly Improve the Legacy Application	Mitigate the major pain points of the legacy application to reduce risk, improve productivity and relieve pressure on the revolution initiative.

Organizing Teams

Team organization is a key aspect in the success of a revolution. An adequate structure helps to keep engagement high and leverages Conway's law to your favor.

Patlet Name	Description
Organize Teams Around the Product Domain	Design the teams' communication paths to reflect the product domain of the company, leveraging Conway's Law to your advantage.
Product Teams Own Architectural Migration	Empower and encourage product teams to take ownership of the migration tasks for the parts of the system they are responsible for.
Dedicated Migration Teams	Assign architectural tasks to dedicated teams working in collaboration with product teams when migration is more complex.
Specialist Migration Team	Create a team composed of architecture specialists to guide other teams in their migration tasks.
Architectural Platform Team	Create a dedicated team to build and evolve a unified platform that standardizes development tools and processes in the new architecture.
Architectural Reliability Engineering Team	Assign a core team to define standards for infrastructure, monitoring, security, and other operational aspects that should apply to all components in the new architecture.
Team Owning the Legacy Application	Have one or more dedicated teams own the legacy application and be responsible for improving it, implementing approved changes, and collaborating with migration teams.
Architectural Retrospectives	Conduct regular retrospectives with the technical and business people within the organization to discuss the new architecture and get insight on the challenges teams are facing.

Other Related Patterns

This paper also mentioned a related pattern from another set of patterns outside these patterns. The following patlet briefly outlines the gist of that pattern.

Patlet Name	Reference	Description
Canary Deployment	[Yoder et al]	Deploy the change to a limited number of users or servers to test and validate the release.