Pega modernization

Unlocking value by tackling technical debt

The business issue

technical debt. One of the most significant challenges we face is that technical debt is not widely discussed and therefore remains a hidden threat. This whitepaper aims to address that.

It is predominantly written for both technical and non-technical audiences who are already working with Pega.

The predominant audience for this is:

- Pega Center of Excellence (COE) and Innovation Hub leads
- Pega product owners
- Pega developers

It is designed to offer thought leadership with opinions, which some may not entirely agree with. The goal is to spark ideas and conversations regarding the impact of technical debt and how it is understood, accepted, and managed. It is fully acknowledged that the balance between technical debt management and feature development will always be unique for each client.

The style is designed to be conversational. The views provided are purely those of the author, with grateful advice and edits from experienced PSAs, technical architects, and enterprise architects – both within Pega and its valued partners.

TL;DR (Too long; didn't read)

Technical debt can be thought of in the same way as credit card debt. It's a reality of life and a useful tool in helping us achieve what we need to. In fact, just as the prudent use of a credit card can boost your credit score, there are many situations where implementing technical debt is helpful and adds value. However, when mismanaged, it's a different story; technical debt (just like credit card debt) can soon spiral, with compound interest dragging you down and preventing progress.

It is a reality of any enterprise development. Whether we like it or not, it's there and the worst thing we can do is pretend it doesn't exist. The more technical debt, the more likely it is to include negative elements that systematically degrade the value:

- More time and money are spent managing the technical debt, so less investment time is spent on delivering topics that directly enhance the organization
- Significantly increased costs (just to keep the lights on)
- More bugs and stability issues
- Increased threat landscape and associated security vulnerabilities
- Decreased usability and worsening user experience
- Major increase in the time spent on upgrades/updates

Not everyone agrees on the seriousness of technical debt. To some purists, all technical debt is bad and should be eradicated. Others take a more pragmatic view and suggest that some technical debt is OK (if not a good thing), as long as it's measured and understood. It may even be used as a conscious steppingstone to a much more strategic outcome. Whatever your view, a universal truth is that not all technical debt is the same: If it's designed with good reason, it's purposeful, measured, and most importantly, short term. But at the other end of the scale, there is technical debt that is inadvertent and used recklessly. In all cases, it's crucial to recognize the difference and manage it accordingly.

Managing technical debt needs a two-fold plan:

Develop a strategy to prevent (or minimize) the accumulation of more technical debt: Supply guidance on expectations, keep up to date with best practices where it is most prudent, and provide governance for remediation.

Manage existing technical debt and attempt to minimize it. Based on our experience, most organizations (who do attempt it at all) tackle technical debt as a "project" rather than taking a strategic, enterprise-wide and ongoing approach.

While searching for what is effectively "bad news" may seem brave, it's all part of a modernized mindset which constantly looks to innovate and optimize business value. Whether we call it technical debt or continuous application quality, it's essential that we gain the consensus of all relevant stakeholders and create teams who have both a "making" and "mending" mindset within their culture.

Technical debt, like credit card debt, compounds over time and, if left unchecked, can quickly become something that stifles innovation and cripples business value.

Use the Best practice checklist on page 9 to start making a difference today.

What is technical debt?

Technical debt (in some circumstances, referred to as code debt) is the existence of any code or configuration that doesn't follow best practice, often a consequence of decisions that prioritize speed of delivery and release over the highest quality of code.

The term is a metaphor for financial debt, where the "interest" is the extra work required, in payment for coding quickly now. Technical debt, therefore, becomes the implied cost of future rework and refactoring that results from choosing an easier solution in the moment.

Within Pega, there are other forms of technical debt. But what is always consistent is that there is debt left behind, which will slow down innovation and degrade the overall value the solution is providing. Some may extend the definition to operating on a version in extended support or, worse yet, out of support. While it is acknowledged that skipping updates will make the process harder and more time consuming, we are purposely narrowing the definition to code and configuration choices, as aligned to the wider industry term.

However, one update-related element worth highlighting is where clients develop custom code for a witnessed bug until a proper patch is available. Our experience demonstrates that this approach (unless reversed) causes significant issues when the patch (and subsequent updates) are implemented.

Sometimes technical debt is used as a steppingstone (minimum viable product, or MVP) to achieve a final strategic solution. This is acceptable as long as there is agreement to remove it when appropriate.

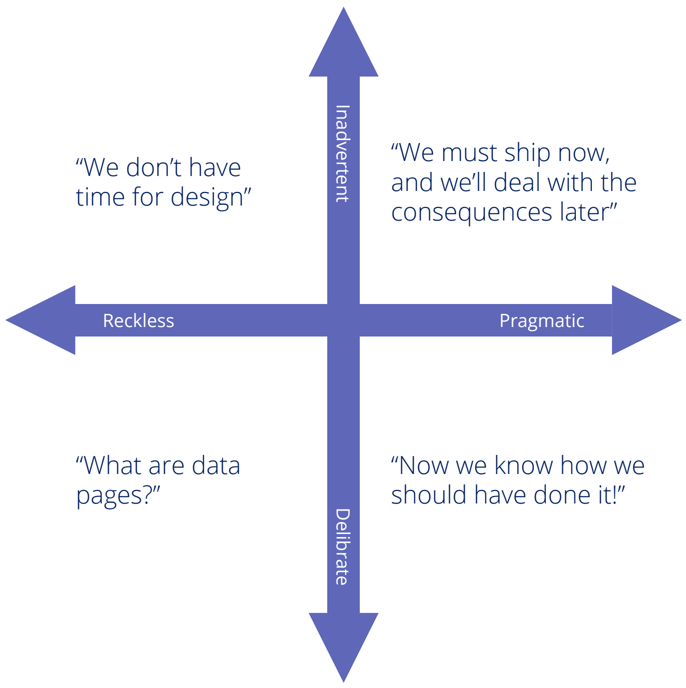
Technical debt within Pega applications

The causes of technical debt within Pega applications include:

- Developing/configuring an application that prioritizes speed of delivery, rather than quality.
- Performing one or more "tolerance upgrades" ("basic updates" in modern Pega terminology) where the platform rulebase (along with industry solutions) is upgraded to a later Pega version, but the application itself is unchanged and works as-is prior to the upgrade. In this way, all changes, enhancements, fixes, and best practices from the new version of Pega are left out and not taken advantage of. This application now contains technical debt: It is not configured in an optimal fashion and is likely to have performance and usability issues, particularly when compared to a quality-assured application that had been factored accordingly. Repeated tolerance upgrades exponentially compound the issue, but due to the slow "bit by bit" nature of the degradation, the issues go unnoticed or, more likely, unreported. For more information on upgrades/updates, see page 7.
- Failing to use new features, standards, and best practices, such as data pages, dynamic referencing, or automated testing.
- Not aligning with industry standard and best practices around DevOps, continuous integration, and continuous delivery. Changes have been made direct to production.
- Implementing custom code instead of using out-of-the-box (OOTB) controls.

In all cases, consider how much debt to carry is appropriate. On one side, you have low technical debt with ease of change and, on the other, high debt that is much harder to change. It's important, however, to recognize that maintaining low technical debt takes a level of ongoing effort.

What is technical debt?


Inadvertent
"We don't have time for design"
"We must ship now, and we'll deal with the consequences later"
Reckless
Pragmatic
"What are data pages?"
"Now we know how we should have done it!"
Delibrate

Consider the following scenarios:

1. To meet regulatory obligations, a healthcare company is required to deliver reports in a timely fashion to satisfy audits. A new reporting feature in Pega helps automate this process (and will be of use to other parts of the business), but this would need implementation. Due to time constraints a simple (scenario-specific) implementation is put in place. This is recognized, accepted, and documented. It will be refactored according to best practice (including promoting reuse) within the next two sprints after go-live.
2. A new feature is needed: While there is no particular deadline, the feature is highly sought. The team manually test the application. They have not deployed automated unit testing (AUT) because the development team were trained on Pega 6 and did not understand the latest ease and capability. Consequently, no unit testing is deployed, each ongoing change needs manual testing, testing isn't comprehensive (due to manual errors), and user experience suffers.

Both of these scenarios create technical debt, but clearly they don't have the same level of impact. The technical debt in Scenario #1 is purposeful, understood, and most importantly, temporary. According to the scale above, it is pragmatic and purposeful and will be addressed in a short time period.

Scenario #2, however, is inexcusable. It comes from a point of ignorance: a technical team who are not versed in the latest capability. It's an example of bad quality development. Interestingly, the impact may not be too visible initially but nevertheless, the impact is real and will expand. Every change, every update will take longer and longer. Updates are then considered slow and may be avoided, impacting the business. Consequently, security vulnerabilities are numerous and benefit-driving features are not available. Above all, the organization's value has been severely impacted.

Not all technical debt has the same impact. When considering technical debt, decide where it is on the above quadrant.

Secret #1: The invisible threat

Someone recently contacted me with some negative feedback: They had seen me present to a client on modernization and were disappointed with one aspect of what was communicated. Their view was: "We shouldn't be talking about technical debt."

When questioned why, he said it would be perceived as a "criticism": an implication that either the client or the partners who developed the application had done a bad job. While this is a reasonably common view, it is disappointing and can become part of the problem. If we can be aware of technical debt, acknowledge that it both exists and has an impact, we can begin to tackle it. This is just one of the reasons why it is essential that this subject is made more visible.

Technical debt is often the invisible threat: It's lurking in your applications, it's not immediately obvious where it is, and the impact is not always instant. But it's there and it's likely to be hurting you even if you don't yet feel it. Whether we like it or not, technical debt develops over time: A "perfect" Pega 6 app built in 2010 will have lots of technical debt today if nothing has been updated.

So, let's be open: Technical debt exists, and it's likely to be having an impact. And that's not a criticism; it's a fact. The only blame should be where it's being ignored and/or not being addressed.

And there may have been justifiable reasons that the technical debt was introduced in the first place, such as:

- Perhaps there is an urgent release where speed is purposely but temporarily prioritized over quality.
- A business requirement required an application to do something, which, at the time, Pega couldn't do out of the box. Now, several years later, this feature does exist in the platform, but the custom-made code is still there.
- Or very commonly, something that was best practice five years ago is simply no longer best practice today.

In summary, we need to stop technical debt being treated as a secret; we need to openly talk about it, and we shouldn't see its existence as a criticism. If, however, there continues to be strong resistance to the term, consider framing the subject as "continuous application health" or a similar phrase that sits better within your organization.

Secret #2: MLPs and technical debt

The phrase minimum lovable product (MLP) or the more generic term minimum viable product (MVP) is a frequently used process to deliver value quickly. To quote from Pega Academy, "Pega uses the minimum lovable product, or MLP, to define a scoped set of deliverables that lead to an implemented use case that delivers value."

It then goes on to state "Future MLP backlog development is prioritized based on Pega and customer strategic needs."

However, we need to surface another widespread secret here: The MLP approach is almost certainly going to result in some form of technical debt. The completely understandable desire to get results quickly is probably going to end up in less than optimal development patterns. This is absolutely fine as long as the trade-offs are understood, documented, and resolved at a suitable time.

Secret #3: Can Pega be perceived as technical debt?

This statement is purposely controversial, and the answer should instinctively be: “Of course not.” However, there is a real risk here. Within this whitepaper we have attempted to surface some of the secrets that people like to keep hidden. This question is perhaps the most unthinkable: Is it possible that the entire Pega deployment within an organization is perceived as technical debt?

The word “perceived” is important here. Because while the Pega team (the COE or Innovation Hub) may believe everything is OK, other parts of the organization may think differently. It may not be said openly, but unless we guard against this, we risk severely impacting the value of Pega, making Pega less tactical and relevant.

Consider a simple solution where an organization has a product API and a customer API. The wider business needs to have a view of which customers have which products. Can this be done in Pega? Of course it can. A helpful Pega team simply runs a data transform to parse this list into a new data class and report definition, which runs for months and months.

Instinctively, you may ask what’s wrong with that? Well, it depends who you ask. To the Pega team (and the original business requestor) it is probably fine, but to the wider enterprise architecture this may not be an appropriate solution to the problem.

Another view would be that this should have gone into the organization’s data warehouse, from where the strategic reporting solution would have been able to create the view. This could perhaps be rolled up into higher-level reports, with subtly different views depending on stakeholders and modernized along with other strategic reporting views.

So to those within the client that subscribe to the view, another (non-Pega) solution should be providing this; the earlier Pega report solution is technical debt. That report (and anything else that follows), and no doubt numerous other examples, are tactical and will require ongoing effort to keep modernized. This risks the wider organization becoming frustrated with the Pega solution, seeing it as a form of shadow IT and perceiving Pega as less strategic than it should rightly be treated.

Just because we can do something, doesn’t mean we should.

The summary here, is to go into situations with your eyes fully open. Sometimes a legitimate solution is one that is pragmatic, tactical, delivers business benefit quickly, and has some acknowledged debt. But equally you should guard against solutions where (often silently) other stakeholders perceive Pega as being the debt. While we all know that Pega can do anything (within reason), that doesn’t mean we should take that on if it’s not in the organization’s overall best interest and doesn’t follow the client’s enterprise architecture strategy.

Pega Platform leads (from the COE or Innovation Hub) are encouraged to ensure they are not simply providing guidelines for within the Pega Platform but steering its use in line with the client’s overall strategy. In line with many other points made in this guide…

- While it might be another unthinkable secret, it’s important we acknowledge the risk that Pega is externally perceived as technical debt and take action to mitigate it.
- Appropriate governance and leadership are needed, not just on the quality of developments within Pega but its strategic and holistic use within the client. Just because something can be done within Pega doesn’t mean it necessarily should be.

Tolerance updates vs. compliance updates

To upgrade or update: What's the difference?

While both terms refer to enhancing the software you're using, the term "upgrade" typically refers to a less frequent, more drastic change. Since the launch of Pega Infinity in 2018, we have worked hard to minimize any impact of these enhancements. Pega highly recommends you adopt the latest version as it becomes available. As such, these should be considered "updates" not "upgrades." A move from version 7 (or earlier) may still be considered an upgrade, but will need updates from that point forward.

Tolerance (basic) updates

- Tolerance updates are a common source of significant technical debt.
- A tolerance update is where the platform rulebase, along with industry solutions, are updated to a later Pega version, but the application itself is unchanged and works as-is prior to the enhancement.
- In this way, all changes, enhancements, fixes, and best practices (from the new version of Pega) are left out and not taken advantage of.
- The applications now contain technical debt (and often significant amounts of it): It is not configured in an optimum fashion and is likely to have performance and usability issues.
- Repeated tolerance updates exponentially compound the issue. But due to the slow "bit-by-bit" nature of the degradation, the issues may go unnoticed or (more likely) unreported.
- Through a tolerance update, the solution introduces two types of debt: "feature debt" (not making use of the latest functionality and features) and "code debt" (moving away from current best practices).
- The impact includes business-facing capability but also within-Pega features that promote speed, stability, and usability. Good examples would include the use of data pages, and dynamic referencing.

Compliance updates

- Compliance updates are the recommended option for performing version enhancements.
- Here the application is remediated or refactored to take advantage of the latest features and best practices.
- Examples would be:

  To adopt the latest UI standards and replace custom code with the use of out-of-the-box Pega controls.

  Convert old or legacy portals to new, responsive portals.

  Using new features and best practices like data pages and dynamic referencing wherever applicable.
- Importantly the remediation/refactoring does not all have to be done in one go, prior to the update going live. The application may be updated and go live using tolerance/basic update techniques, but remediated in the following weeks/months as appropriate (the "Evolve" phase).

What’s in it for me? Seeking support from everyone

Setting arbitrary rules rarely works. In terms of the Process Technology People triangle, we will not be successful if we don’t get support from all the stakeholders involved. This engagement is just as important, if not more, coming from the individual developers as it is from the leadership team.

A plethora of technical debt is bad for everyone and has an impact across the organization:

Engineers/developers

- It’s less likely that you will get to work on the latest (more exciting) capabilities.
- Making a difference (adding value) is much more difficult.
- Fixing bugs is more challenging.
- You become bored – each day is the same and work becomes a chore.
- You don’t want to face another (painful) upgrade.
- Other job opportunities seem appealing.

Development teams

- Lower velocity
- Significant variance in velocity.
- Updates/upgrades are painful and avoided.
- Making and agreeing to plans (that you’re confident in) becomes much more difficult.
- Team morale suffers (with high turnover).

Wider organization

- Much higher time to value.
- High proportion of budget spent on maintenance (“keeping the lights on”) – and consequently, less on business objectives.
- Legislation and compliance are hard to achieve.
- Software versions are old (significant caveats and vulnerabilities).
- Customers and end users have a poor experience.
- Less reliable response to customer issues.
- High friction between business and IT.

Beware of a trap where “fast” individuals are considered the superheroes and get to work on feature development, whereas others (who are equally capable) are purely there to fix what the first person did (the technical debt that was created). This can be very demotivating for the equally capable developers who are doing a fantastic job. Menders are superheroes too!

Best practice checklist

Whether you have a traditional COE, an Innovation Hub, or one or more Pega Partners or in-house developers providing solutions, the only way to optimize your Pega investment is through providing strong guidance/enforcement that aims to acknowledge, understand, and (where appropriate) prevent technical debt.

People

- First and foremost, ensure that the developers (both in-house and partners) understand and agree on the need to manage technical debt. All stakeholders need to be on board, but none are more important than the developers themselves.
- If the phrase "technical debt" is too negative, consider using "continuous application health" or a similar alternative.
- Evidence suggests that in most Pega development teams there are those who gravitate more to "feature" work and others to "fixing" work. These so-called "makers" and "menders" are equally important – build a team with individuals who enjoy doing both and who are championed as equally important within the organization.
- Present the value to the whole organization. While the details may not be relevant to everybody, all stakeholders should buy-in to the benefits of quality development.

Governance

- Turn addressing technical debt into a business as usual activity. Whether a daily (or at least per-sprint) practice, make this the default behavior and part of the organization's culture. The non-functional backlog should not be considered less of a priority than the functional backlog.
- Prioritize technical debt in the same way as financial debt. First "staunch the bleeding" and "cut up the credit cards." Teams cannot dig themselves out of technical debt if they continue following the same behaviors and practices as before.
- Ensure appropriate governance within Pega practice, whether that be classified as a COE or Innovation Hub. Where no such practice (yet) exists, the subject of technical debt is just one good reason it should be considered.
- Reuse is an essential Pega foundation, and the failure to develop features as reusable components (when sensible) should be considered technical debt. Have clear guidance as to which components are (or should be considered) reusable that already exist in your deployments. Ensure such components are in a specifically designated area of the Situational Layer Cake™ and that a suitable catalog exists for use, ownership, and maintenance.

Discover

- Do you have a good understanding of the technical debt currently within your applications? Use metrics that indicate when and where your team is being slowed down.
- Clients are advised against simply executing a "technical debt removal project." While an initial project will add value, technical debt management should be seen as business as usual and become part of the ongoing modernization adopted within the wider organization.

Prepare

- Check that your developers are skilled in the latest Pega versions and best practices. You may have the cleverest LSAs and PSAs who bring many years of experience. But if they completed their training in version 6.x and haven't kept current on modern best practices, there is an increased likelihood that more technical debt will be introduced. This should include industry-standard skills such as DevOps and Security.
- When utilizing partner developers, ensure they are comprehensively trained on the latest versions and best practices. Ensure both partner managers and individuals are fully versed in and agree to your technical debt–focused governance. Where appropriate, seek guidance on the partner's technical debt guidelines to ensure they are aligned with that of your organization.

DevOps

- Strictly use DevOps pipelines to build and manage your applications. Block any and all direct access to the production environments so fixes can follow DevOps principles.
- Regardless of how you deploy it, you need full automation. Full automation means the ability to recreate the application from an artefact repository – preventing incremental environmental difference, where personnel are manually editing product rules ("build file") that will likely lead to (hidden) technical debt.

Get started quickly

Crushing complexity: Bringing business and IT together

Pega Express™ is a light, design-focused delivery approach that uses Pega's low-code experience, best practices, and scrum to deliver meaningful outcomes quickly. Pega uses design thinking techniques to break customer journeys into smaller, more manageable pieces called Microjourneys®. The main Pega Express phases are shown here:

- **Discover** innovative solutions to your business problems.

  Design thinking principles and identify Microjourneys, Personas, Channels, Data, and Integrations.

  Identify microjourneys and prioritize them for iterative delivery.
- **Prepare** and define your Minimum Loveable Product (MLP) by Directly Capturing Objectives (DCO)

  Capture and priotitize user storeis using Agile and Scrum tools
- **Build** applications by using business confurations in the platform.

  Collaborate with business using continuouse feedback loops, show and tell sessions and automated testing tools.
- **Adopt** your Minimum Lovable Product by incrementally going live with business outcomes.

  Measure success whilst planning your next MLP.

While we highly recommend using Pega Express as a methodology, clients are also advised to complement this with actions that manage technical debt. A tried-and-trusted method is to ensure technical debt is treated as stories within existing projects that add business value.

This aligns to the following phases:

Discover

- Identify what elements of technical debt already exist in your applications.
- Identify personnel who naturally gravitate to being "makers" or "menders."
- Check the team and partners are trained in latest versions and best practices.

Prepare

- Ensure everyone (particularly developers) are aligned.
- Build teams with a mix of "makers" and "menders."
- Ensure suitable governance is in place.
- Identify where the MLP model will create technical debt and plan for remediation.

Build

- Ensure suitable time for technical debt remediation is part of the culture.
- Design, identify, and promote reuse, and implement whenever appropriate.
- Strictly enforce DevOps, Agile, scrum tooling, and best practices.

Adopt

- During deployment, ensure latest capabilities are being adopted.
- Look for new opportunities for adoption in subsequent phases.
- Identify when rapid MLP launch results in technical debt. Document and add fixes into subsequent sprints.

Technical debt examples

Custom level infrastructure – An insurance provider from North America had configured direct database access from their application. The application was severely inefficient as well as causing delays during updates.

Solution: Refactor away from custom-level infrastructure

No automated unit testing – A pension provider in Europe had not deployed any unit tests through their applications. This was not apparent in the day-to-day use of the application, but platform updates became increasingly expensive, prolonged, complex, and eventually, avoided.   Consequently, the business value was adversely impacted, and user experience was poor.

Solution: Provide up-to-date training, starting with automated unit testing. Update urgently and remain current.

Custom GDPR control – In 2015, an organization in Europe was concerned about the impending introduction of the new General Data Protection Regulations (GDPR). So they implemented a custom method to provide compliance. Now, a standard OOTB Pega capability exists for this task, CBAC. Many years later, analysis showed very large heap memory usage with hundreds of instances referencing the same data page. Responsiveness was extremely slow, and user experience was poor.

Solution: Refactor application using OOTB capabilities

Accelerating through robotics – At a European government organization, a connection to a system of record was required, which, at the time, had no defined API. In order to accelerate the value of robotic process automation (RPA), attended RPA [also known as RDA] was used to capture the necessary data. The third party's development team had a significant backlog and was unable to build an API immediately. The use of RPA therefore became deliberate, purposeful technical debt used as a stepping stone to strategic deployment. As soon as the API was completed, the RPA (and therefore the associated technical debt) was removed.

Summary

Whether we call it technical debt or "continuous application health," it's an essential subject affecting agility, innovation, and ultimately, value.

- Technical debt exists and discussing it is not a criticism. The only criticism should be pretending otherwise.
- The MLP/MVP delivery method is great at delivering value early, but we need to be careful not to over prioritize agility over quality. When an MLP has been delivered to provide early value, the associated technical debt should be documented and addressed when appropriate.
- Make deployment choices that are in the wider best interest of the organization. Tactical choices made without context in the wider business risk Pega being perceived negatively, with larger consequences that dwarf any benefits seen.

Final thought... Consider how much technical debt is appropriate to carry. On one side you have low technical debt with ease of change, on the other is high debt that is much harder to change. The most important thing to do is to start somewhere and make managing technical debt a conscious decision that is weaved into your development culture.