



Over the past 6 years, APR has assisted a number of clients in moving to hyperexponential's pricing decision intelligence platform, hx Renew. In this document, we talk about some of the lessons we've learned from both a technical and a practical resourcing perspective that you might want to consider when planning your own hx Renew implementation.

Programmers or actuaries?

It's worth carefully evaluating which tasks in model development require specific domain knowledge. With a well-specified methodology and / or a well-documented existing model to work from (ideally both!), developers without pricing experience may be able to carry out much of the day-to-day work of coding the models in hx Renew.

Division of responsibilities in this way drives efficiency by allowing each group to hone in on the areas where they can add most value; pricing actuaries take the lead on important model design and rating methodology decisions, while developers can focus on 'translating' these designs into robust, well-documented and efficient code.

A total division of responsibilities is not always appropriate, and a successful development will likely require at least some participants to appreciate 'both sides of the coin'.

Upskilling of existing staff



Upskilling of existing staff – that is to say, training experienced pricing actuaries to be skilled programmers, or vice versa – is one approach that can be taken in resourcing a hx implementation. Indeed, hx Renew's user-friendliness makes this very much doable, albeit dependent on a 'baseline' level of familiarity with coding in Python. (Fortunately, the team at hyperexponential do provide basic Python training to help customers get started). In the longer term, this approach has the desirable outcome of allowing actuaries to take full 'ownership' of both their models and the underlying code.

However, timelines can be dragged out significantly if training needs to happen prior to, or in parallel with, development. Furthermore, moving actuaries to full- or part-time development has an associated opportunity cost, and risks draining resources away from maintenance of existing models or other important BAU tasks, such as case pricing.

Instead – as alluded to in the previous section – consider this as part of a blended approach, where more experienced coders work alongside more experienced pricing actuaries, each group learning from and supporting the other.

Plan your models holistically

In the early stages of a migration project, it's important to think about the 'big picture' before diving into the specifics of any one model. What does it look like to build a model? What is the development process, the testing process, the release process, the definition of done? Take the opportunity early on to establish best practices, coding standards, and 'house style'; these provide clear guidelines for developers to work to and ensure consistency across a codebase. It is much easier to work in a prescribed style than to impose one retroactively.

Consider how to keep your models 'aligned' with each other – naming conventions and data schema is a big one here – but don't force the fit. Take the opportunity to identify patterns, common data items, and 'chunks' of functionality common to multiple models, such as rate change or connections with external systems. Earmarking these allows them to be given appropriate attention, aligned stylistically, and prevents unnecessary reduplication of effort. In some cases, they may provide opportunities for code reuse (for example, in a shared library), but be wary of premature abstraction - trying to force models into a 'cookie-cutter template' is rarely the correct approach, and over-sharing of code can create a tangled web of interdependencies that is difficult to unpick.

Be careful of scope creep

It's also important as part of this planning process to not just decide what you intend to build, but what you don't intend to build. Deciding what functionality is in and out of scope of the initial development window ensures that everyone is clear on success criteria and reduces the risk of failing to meet timelines. Establish a clear distinction between 'core functionality' - the things that your models need to be able to do – and 'peripheral functionality' / nice-to-haves.

Try where possible to avoid slipping new features into the migration process, which can distract from the primary objective of hx Renew implementation. Instead, focus on replicating the existing model, and add these features into a longer-term plan where they will be implemented in a future phase of development. Don't be afraid to develop iteratively, and don't be afraid of getting things wrong; consider whether a 'cadence' of regular model releases may be more appropriate than a 'one-and-done' big bang approach.

The first model release is about establishing a foundation for future releases to build upon; keeping the scope tight helps ensure this foundation is as sturdy as possible. There's a lot to get excited about with hx Renew – machine learning, process automation, web-scraping, to name just a few – enabled by a combination of access to Python libraries, APIs, and integration with internal and external services. However, the underlying model needs to be sound before these can be effectively enabled; in many cases, these tools are only as useful as the pricing data that is fed into them.

Remember your connectors

hx Renew and your suite of pricing models are only one part of a larger ecosystem that involves actuaries, underwriters, IT, data teams, and beyond. Data needs to get into the model in a useful way, and data needs to get out of the model in a useful way. Think carefully about any policy admin systems that you intend to use, any data capture you want to do, any workflows that pricing data is involved in, and ensure that the relevant IT and data teams are looped into the planning and development processes as required. Good data structure supports future reporting and MI, and working with data teams is the best way of making this a reality.



Engage with underwriters early and often

Fundamentally, underwriter buy-in is at the core of a successful hx Renew model migration; you can build the best pricing models in the world, but it makes no difference if your underwriters won't use them! Ensure underwriters understand the reasons behind the migration, and involve them in conversations about design choices which directly impact them.

The sooner you can kick-off the feedback loop between developer and underwriter, the better - while showing off an early or unpolished prototype might feel 'embarrassing', getting early feedback can be invaluable. (Issues flagged a month before release, while the codebase is in a more 'mutable' state, are much easier to fix than those flagged a week after).

What's more, working with underwriters to build a clear, linear workflow - a model that is easy to populate 'correctly' - maximizes the likelihood of correct model use, resulting in better and more consistent data capture.

hx Renew is not Excel

When moving from one modelling platform to another - for example, from Excel to hx Renew - a good starting point is to replicate the rating logic in an existing model as closely as possible in the new software. This is eminently sensible - it allows for like-for-like regression testing simply by running the same inputs through both and checking that the outputs produced are the same. However, it's important to recognize the ways in which Excel and hx Renew differ from each other when making design decisions.

For example, it may not be possible to recreate your Excel spreadsheet's layout and colour scheme exactly in hx Renew. Consider this freeing! In Excel, the flow of calculations is inextricably linked to the layout of cells, which can limit how models are structured. hx Renew separates the user interface from the calculations, allowing for cleaner, more intuitive model design. You don't need to copy Excel's layout or limitations - use hx Renew's components to create a smoother workflow for underwriter.

hx Renew also supports more advanced features like Python libraries, asynchronous tasks, and automated testing. These open up new possibilities that Excel simply can't match. So, while it's helpful to start with what you know, embrace the differences and take advantage of what hx Renew can do.

The two T's - testing and training

A hx Renew model isn't complete until it has been thoroughly tested to ensure that it's operating as expected. This may involve unit tests of specific model components, regression testing against a previous model, or end-to-end testing including any other connected data platforms. It's also important to test that the model is both usable and user-friendly; for example, this could include testing that the interface is clear and easy to navigate, checking that the rating algorithm runs in an acceptable time frame on large data sets.

Similarly, make sure to make sufficient allowance for training into your implementation timeline. Underwriters will need to be trained on how to use the models to price risks, flagging any changes to process or differences from the previous models; consider how, when and by whom this training will be delivered. As well as ensuring a smooth transition from the old to new models, if timed appropriately this can be a great way of layering in some additional user experience testing, gathering feedback on how the models feel to use and any pain points in the underwriting process.

Be careful with post-release bandwidth

When your hx Renew models have gone live - including testing and training - it will naturally make sense to 'taper down' the number of hx Renew developers. However, it's important to remember that there may still be a number of 'business as usual' tasks associated with the model for which developer resource is required. At a minimum, you'll need to make sure you have a plan for raising and fixing any identified errors, providing tech support as underwriters 'settle in' to use of the new model, and updating bases and parameter tables on a regular cycle.

If there is any further functionality to be developed on top of this, it's important to resource this appropriately. While this may not require the same number of developers as the initial migration, under-allocating developers at this point is likely to end up with developers torn between the priorities of day-to-day bug fixes and new feature development, hampering their ability to make efficient progress on the latter.

Ultimately

Ultimately, a hx Renew implementation project is like any other: success depends on thoughtful planning, clear timelines, and balanced resourcing. But just as important is the mindset you bring to it. Don't be afraid to get things wrong the first time - mistakes are part of the learning process. The ability to pivot, adapt, and respond to feedback is often more valuable than sticking rigidly to a plan that no longer fits. Build with confidence, but stay open to change - that's what sets up long-term success.

Want to find out more?



James Nicholl

General Insurance Lead
james.nicholl@aprllp.com



Jacob Warbrick

Senior Actuarial Associate
jacob.warbrick@aprllp.com

Further Information

Further information on our General Insurance Proposition, including case studies, can be found on our website:
<https://aprllp.com>.

