Intro

- Traditionally quant analytics are built aligned with asset class verticals, according to how trading desks are organised. This keeps them close to business and responsive to business verticals' needs. Deployment is typically aligned with project needs, again vertically organised.
- There are typically millions of lines of legacy analytics code in C/C++ and exposed through shared libraries.. With good reasons - it is transferable, it makes writing fast analytics easy, it is well understood, and automatic optimisations are very advanced (because of how low level these languages are and because OSs are written in C).
- Pressure to modernise is many-fold:
    - Regulatory pressures mean that horizontal views of institutional exposures are necessary
    - This in turns provides new capabilities and new views of risk - opening up new opportunities for cross-asset structures trading and risk management
    - ML developments promise faster\cheaper modelling and new, powerful, analytics, and insights; better service to desks, risk management, and external clients.
    - This opens up new business opportunities with analytics as product/service
    - Explosion in technology advancements provide both the pressure to stay competitive in facing clients (internally and externally) and ability to gain competitive edge through highly specialised development.
    - Time to market, productivity of scarcest resources, business innovation, and resource retention all suffer hugely.
- Everyone is trying to modernise to take advantage of all this and stay competitive, but it hurts. It hurts everywhere and it hurts everyone. It hurts a lot, and it really shouldn't, many companies outside of the world of finance are doing all this and more. Why does it hurt so much?

- As a matter of course, an analytics library these days needs to be deployed:
    - as an Excel add-in (for speed of testing and development, for minimising time to traders' desks when it matters)
    - as an in-memory component for C++, Python, C#, Java ... GUI driven applications
    - as a part of a standalone service deployed on-prem for cross-business access to analytics
    - as a containerised micro-service on a cloud for cross-business access to analytics
    - as a service back-end to modern web technologies based GUI applications (either a standalone service of a containerised scalable service on a cloud)
    - as an executable component for large QA runs and CI/CD tests

    - all this while taking advantage of an exploding list of new technologies: cloud, availability of specialised hardware (GPUs, FPGAs, specialised storage ...), modern GUIs, SaaS, shift towards platforms other than PCs, LLMs etc

- At the same time, these libraries are always reliant on large amounts of reliable data, in Dev, QA, and Production environments across deployment environments.
- When production breaks happen in any of the environments or deployment modes, debugging and fixing them is urgent.
- Regulatory obligations are to be able to recreate entire environments, code and data for years into the future.
- Competition pressure makes time-to-market highest priority.
- Innovation falls behind, boredom and frustration become the name of the game; resources exodus is common, recruitment is difficult. Competitive edge of the business suffers.
- Providing all the talent to keep all this going and moving ahead within focused quant teams has become infeasible.

- While deployment modes and data management are steaming ahead and branching independently, core financial analytics remain and will for foreseeable future remain low level and focused on scientific coding.
- In the tech world, processes, specialisations, and powerful infrastructure tooling has been developed to support dev life-cycles within the complex and demanding new environments. DevOps, DataOps, ModelOps, dedicated GUI teams, Data architects ... are all separate professions for a good reason.
- However, these are all focused on the life-cycle of small incremental improvements and quick deployments, typical for innovative low-risk tech based deliveries.
- Fin-eng analytics rarely fit into this framework:
    - large legacy code base, often hard and slow to build;
    - models take months to develop and more months to test, validate, and reconcile;
    - even simple improvements and fixes often have to go through extensive optimisation, validation, testing, and reconciliation cycles;
    - innovation is often necessary to stay competitive and may require further extended development;
    - changes are driven by market forces, regulation, and competition, and are difficult to impossible to plan ahead.
    - priorities are dictated by trading strategies and must be responsive to frequent or sudden changes
- In addition, migrating businesses to new technologies is extraordinarily difficult, because there can be no breaks. This refers to both time and consistency - sudden changes in output analytics are not allowed. Reconciliation costs are becoming huge.
- At the same time, quant groups are the most difficult to organise, run, and recruit, as well as most expensive. Quants are the scarcest resource in a financial organisation.

- In the past decade or so, attempts have been made by very large institutions to address this by rebuilding entire internal code bases from scratch in global monolithic frameworks with IT being run in the vain of huge tech companies. These take many years and cost billions, and include commitment to a particular architecture that will become outdated before the work is even close to completion. The cost is incredible, migration is very painful, talent retention is low and based on pay levels; quant productivity is at abysmally low levels.

- We propose a fundamentally different approach focused on enabling smooth integration between teams focused on distribution of responsibility that is by now well established in the tech world.
- The core principles of the approach:
    - Isolating core modelling from the complexities of deployment and data access.
    - Providing a comfortable, smooth, efficient, and simple standardised development/QA environment. Dev life cycle complexity should not impinge the flow.
    - Ensuring consistency between development and production environments by default.
    - Ability to support disorganised and not synchronised migration across business verticals while delivering consistent modelling and analytics.
    - Ability to support many stages in migration evolution at once.
    - Feasible and efficient distribution of responsibilities between teams, and smooth interfacing between them.
    - Reducing infrastructure development duplication.
    - Empowering analytics developers to engineer and innovate with confidence about deployment complexities being handled correctly.
- We do this by wrapping analytics into cushy layers of standardised environments, processes, project templates, and boilerplate code that make it easy to integrate them into other environments and processes with confidence, while not limiting access to advanced features when they are needed.

Interfaces

Quants should not need to worry about interfacing technology.
All they should do is specify the details of the external interface to their library, with appropriate annotations for user help.
Everything else should be automatic.

- We focus at first on C++ analytics. This is because they are the hardest to deal with and are more often than not necessary part of the mix. However, the same approach is applicable to any language\tech-stack.
- We shift focus from the technology of interfacing for particular environments to providing coherent APIs that are easily exposed to any language\environment.
- We then automate the building of interfacing layers and provide selection of available interfaces through build configurations.
- The onus is on the design of interface functionality with the particulars of the technology abstracted away. These abstractions are owned by the owners of the analytics.
- Primary focus is on exposing functions - they work well in every environment\language.
- OO interfaces can be provided, of course, either directly where possible, or by using handles pointing to cached objects where not (e.g. Excel, Rest).
- OO structures should be clearly understood as a form of caching for stateful coding.
- We expect to still provide access to specialised functionality within various technologies - this is meant to be empowering not limiting in any sense. Owners of the analytics should be able to utilise most optimal access to technology when needed.
- Such cases should, hopefully, be rare - we need interfacing layers to be good.
- Standardised implementation of interfacing layers will later help automate further processes better as well as provide smoother interfaces to other specialised teams and ease reconciliations.

Excel is a pain to deal with, but it ain't going away.

Ability to quickly develop efficient, complex, working applications that deal with lots of numbers in Excel is second to none.

It is also very familiar to quants and traders.

Even if it is never used as a final delivery target, being able to use analytics in Excel speeds up the long early stages of analytics development, demoing, refinements etc hugely.

Pretty much every house will have a number of implementations of Excel - C++ interfaces.

We propose development of super simple, zero-knowledge interfacing layer focused on principles above.

Later we can use the same syntax for exposing the same functionality to other languages\environments.

Once we start getting serious about deployment in a maintainable fashion and at scale, we start looking at interfaces between languages.

In a well designed development environment exposing functions to Excel or to any other language should look exactly the same.

Specifying that a function is a part of the interface to analytics, providing help text, arguments names and so on, should only happen in one place.

Build configurations should then take care of building appropriate binaries (across OSs).

There are very often problems where underlying analytics need to be further orchestrated in complex ways.

For example, composition of low level functions for computation of risk, describing complex structured products, composing ML models ... anything that requires simple workflow like composition of underlying low level functions.

DSLs are ideal for this and with advances in PEG grammars and packrat parsers are easy to build efficiently. We propose setting up a simple to use framework for building simple DSLs.

Additionally, these internally always provide an implementation layer in the shape of a structured symbol tree. These are in turn representable as pure data structures, which are easily translatable to any structured data format (JSON, XML etc).

Even better, these are easily built with modern GUI frameworks - the tree based structures are isomorphic to data structures of GUI frameworks, and so, we are providing a layer for GUI interfacing as a pure abstraction focused on analytics deliverables.

The next level of complexity is exposing the functionality on the wire.

Sometimes this requires a lot of hardcode engineering where interfaces defined above would be housed and used by a sturdy service, often in Java. These would typically be built and maintained by specialist IT teams.

In the modern world of clouds, the need for that much engineering just to make something available via http is actually rarely justified. More and more often it is enough to build a run of the mill Flask (or similar) based microservice, throw it to DevOps to put up, stick an ingress in front of it, scale it as needed; stay in control of your releases.

Interfaces to this layer can be provided in the same way as all the above ones, automatically hooking into boilerplate code provided at the project setup time.

Integration with Development Environment

The framework of libraries and boilerplate code described above is still code that requires some (albeit small) effort to set up and use. It is not much, but it is still a burden - we can do much better than that. The aim is to get to the state where quants do not have to think about it; zero-knowledge standardisation of API tech.

We can do even better - all this still requires context switching and interrupts the developers' flow. We want to enable engineers to concentrate and flow with the creative work they are doing, not spend time learning our frameworks again and again every time they need them.

We can integrate all this within development environments, provide wizards that will create projects, configure them for particular deployment targets, build them and so on. Much like native project setup and wizards do in any of the popular Dev Envs.

One click and a short walk through a couple of dialogues should be enough to set up an entire new project with all the features above configured and in place.

One click, and a method is exposed, another one, a DSL framework is added etc. Debugging is ready out of the box, in Excel, in GUIs, in REST services, standalone apps.

Builds are portable when the projects are new.

This makes it easier for quants to stay in the flow and brings us closer to standardisation of API technologies.

Portability and Repeatable Environments

Portability is still a problem - the variety in deployment means that code is often not delivered on the platform it is developed on.

This is where nightmares live. Ideally, we want to be able to enable quants to develop/build/test/fix code in the target environment. We also want to be able to quickly recreate the entire development environment once there is a break in QA or Production.. The ownership of the code is not complete before this is in place.

We can do all this with containers.

We setup a hierarchy of standard containers, a prod release one (for cloud releases, obviously, but also mimicking other deployment environments faithfully if need be); then a QA one which contains additional layers for testing and logging., and finally a Development one that also contains additional layers that enable development and debugging within the container.

This way, every project is set up with a set of containers that enable all the three stages from the outset. These are standardised and tested. During every stage of development, a quant can launch a container, build their code within it, debug it, test it and even entirely develop within the container. Goodbye rounding differences, late build breaks, DLL hell, environment configuration differences.

When the code is released, all three containers are released. When something breaks, the entire fully versioned environment can be up in minutes, ready to recreate problems and fix them.. Regulatory requirements for being able to recreate pricing for years in the future are solved simply by storing the containers for long enough.

Once the development is done by quants, they can deliver the containers to DevOps with specifications about maintenance, scaling etc The distribution of responsibilities is simple and natural.

Full Integration

With the container hierarchy in place, we are almost there in the state of nirvana.. Few more steps ...

Firstly, we integrate the container building, launching, and deploying into the development environment.

A click of a button builds the project in all target environments, another click brings up a debugging environment in the target deployment environment (we debug for windows just as easily as we debug for linux within a container basically). Yet another click pushes code to github and containers to whatever container repo is used locally, versioning is automatic. All the boring stuff is taken care of.

With all the standardisation, we can start looking into standardising communication protocols and schemas between services. This can become part of the project templates easily.

On top of this, as part of the environment, we provide standardised integration with Data layers. Either in libraries or as boilerplate code; depending on what works best in the site setup.

We also go big: no single project template will suit all use cases. We set up a framework for building and downloading project templates centrally as a service. There are vendor solutions for this, or we build our own, it's not actually a big deal at all. We should be more focused on the procedures and guards around these project templates than the actual technology for delivering them; proliferation is the enemy here.

These templates are now standardised and well known - DevOps and CI/CD processes can be built around them independently. Assumptions required to make all the nice devops services and CI/CD processes can be built into the templates.

If we are careful when building the templates, we can manage to get them to cover all use cases within two standard deviations of plain vanilla ones. To cater for the remaining 5%, we make sure that the templates are extremely transparent - everything can be overwritten, everything is source and modularised carefully. Code reviews should include justification for using non-standard extensions, but they should anyway.

Hoping for one solution that fits all conceivable problems is unrealistic, but with the ability to add templates, clear responsibility boundaries between quants/dev ops/data ops, layered templates structure, we can cater for most of it. And that is a huge saving.