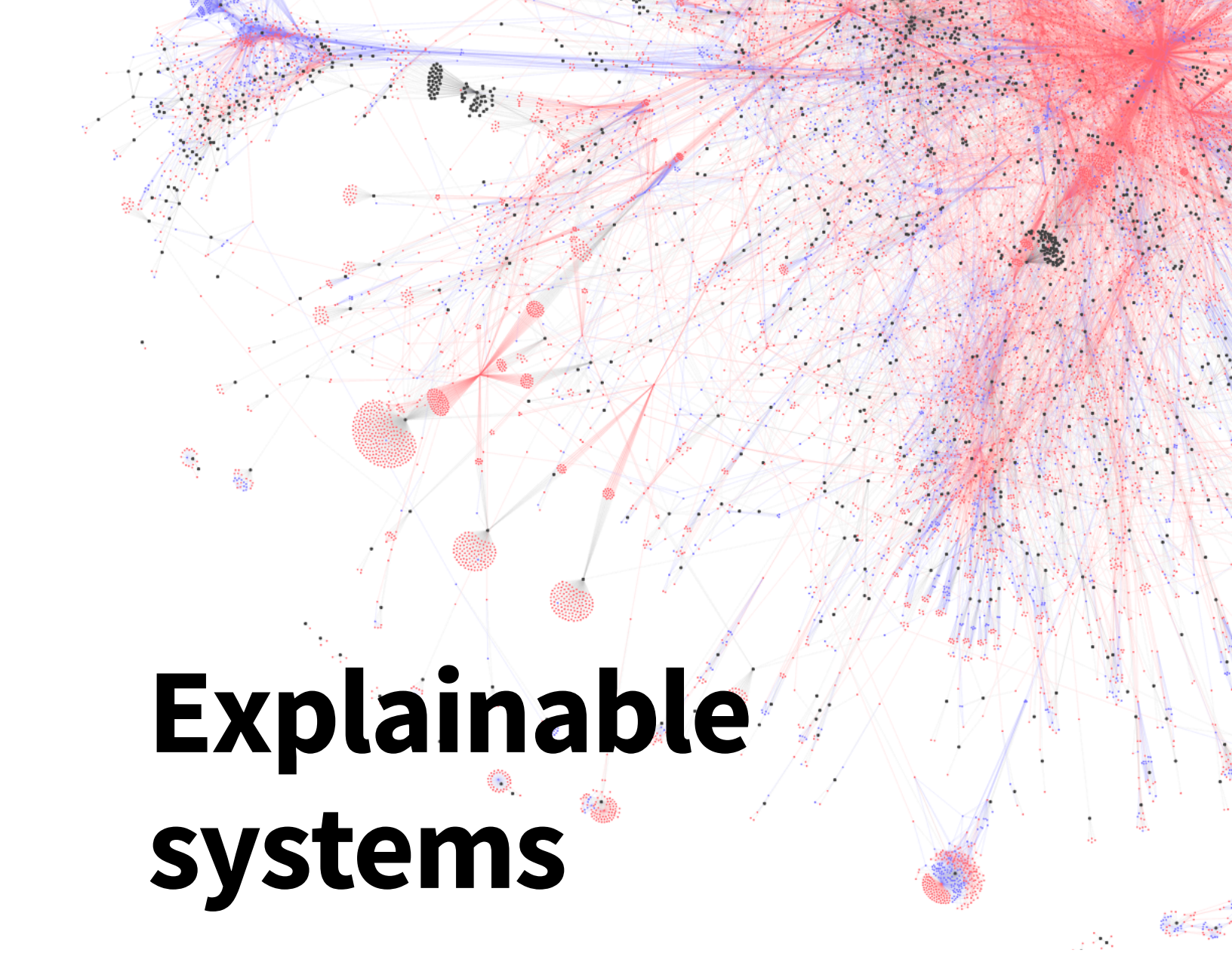


Explainable systems

feenk

The value of your software system today is given by its external functionality. Tomorrow, its value is given by how well you can adapt it. This depends on your ability to understand the system's internals enough to guide its evolution.

The explainability of your systems must become an explicit focus as though your business depends on it. Because it does.

You have a legacy system. And a crisis unfolds.

Maybe, you have just failed a migration. Or, the system's failures appear in the news. Or, you can't adjust the system fast enough to keep up with the market. Or, you just don't know where to start that modernization project.

Often, in such situations, everything looks like an insurmountable problem. That’s a sign of a lack of accurate insight into the system.

You do not lack the ability to solve the problem. You lack the ability to see the problem.

Let’s consider a concrete case. Say you just observed that you you badly need to scale the system. It’s a business imperative. And, as a first step, you decided to split a larger system into smaller parts.

You look at the architecture diagram showing the top level dependencies and decide that, from a business perspective, you’d first like to split the ordering and the scheduling subsystems from one another.

The plan is made. A separate team starts working on the split. Yet, six months in, it feels like the effort is moving in circles. Nobody seems to know whether progress is being made, or approximately how much work is still left to do.

What can the problem be?

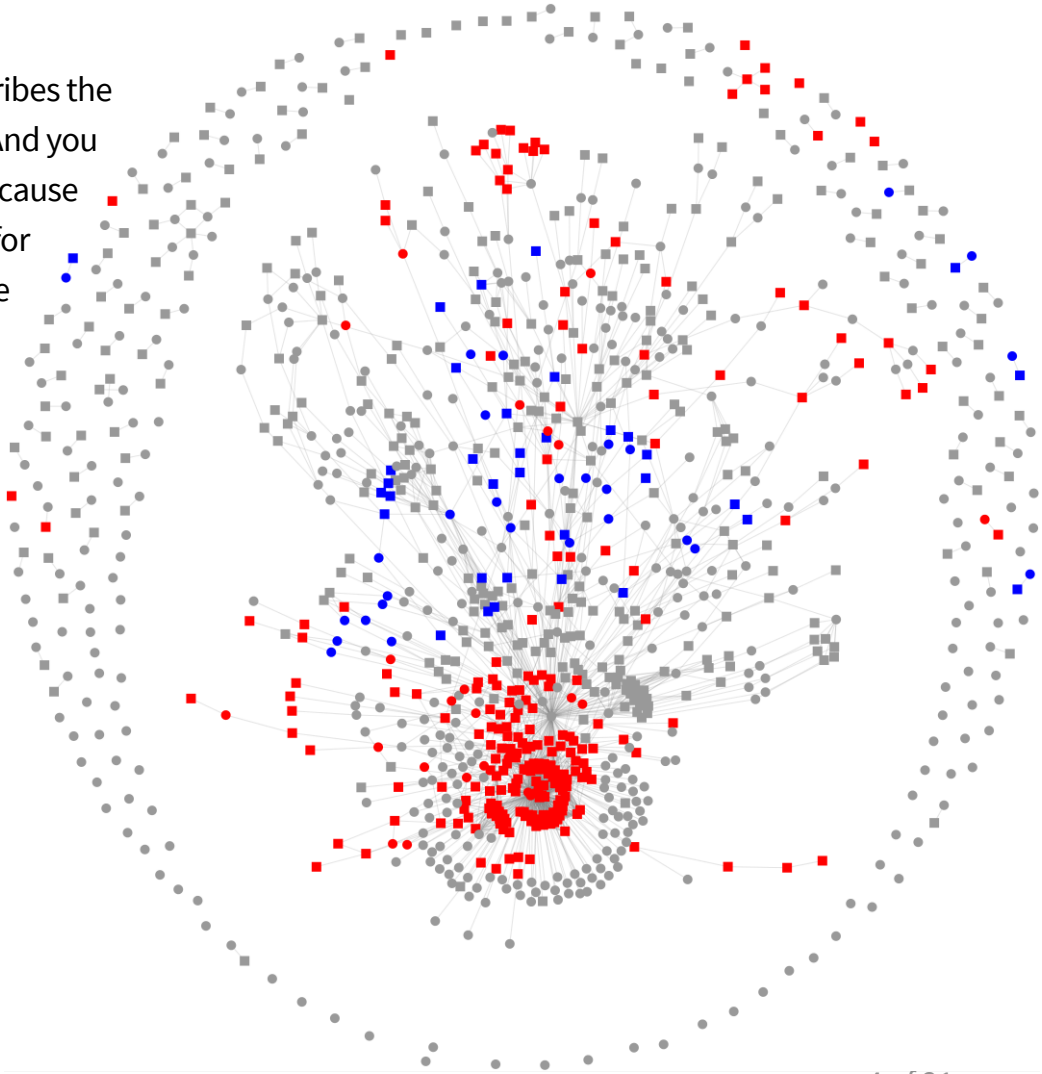
The most important problem is that you do not know what the problem is. When you embark on a business critical project, the lack of visibility poses unnecessary risks that you simply do not need. You have enough challenges already.

Visualizing the system reveals a different picture. You see that the two subsystems have pieces that are highly interlinked (red and blue) among themselves and with the rest of the system. This picture looks very different from the manually drawn diagram you looked at before. It becomes evident that tackling the code in isolation is not the appropriate path.

This picture below describes the nature of the problem. And you now trust the picture because it is created specifically for this problem. It takes the specifics of the frameworks used into account for identifying all dependencies. And it emphasizes the specific entities of interest (the red and blue parts of the two components). If the original diagram was someone's view or opinion, this picture represents reality.

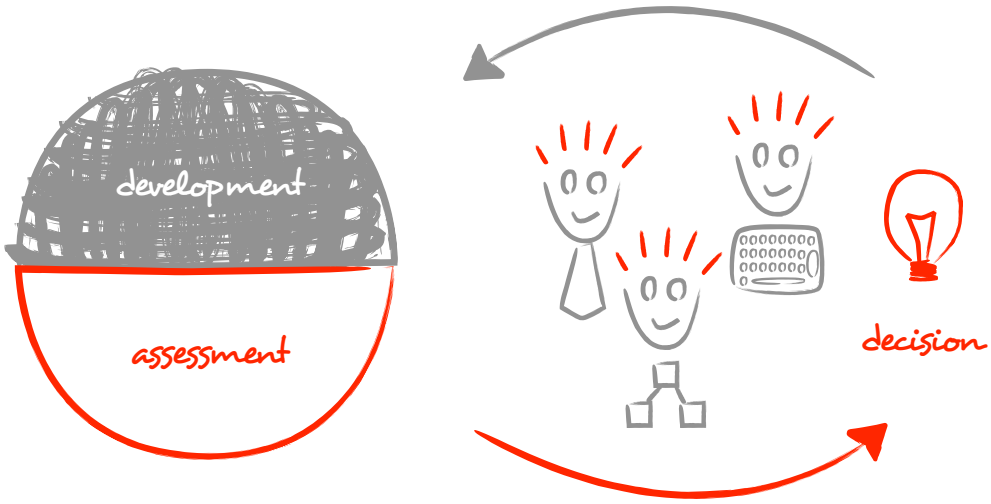
Now the problem has a shape you can trust. You do not need to work from gut instinct. You know.

Of course, this is just one case. Making problems tangible is a skill you want to be available all the time. All facets of a system can be made explainable.



When we think of software development, we often think of the active part of creating the system. Yet, the largest cost is spent on assessing the current state of the system. Developers alone spend 50% or more of their time reading code to know what to do next. These are only the direct costs. The indirect costs can be seen in the consequences of the made decisions.

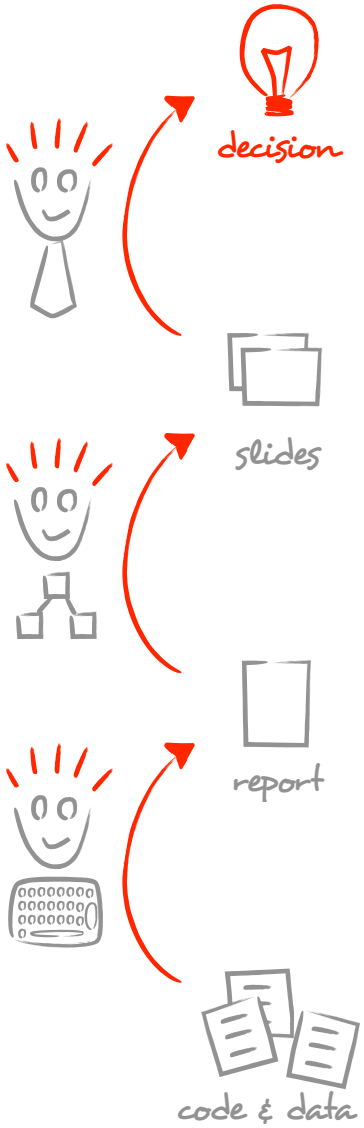
Figuring out the system is the single most expensive activity in development.



Software assessment is the single most expensive activity in software development. Yet, currently, it is not addressed explicitly, and thus, it never gets optimized. Reading is the most labor intensive possible way to extract information from data. Given its costs and impact, this has to change.

Assessment must be approached explicitly.

How explanations are created matters. They are useful only when they relate to reality.



The internals of systems comprise technical issues. Shouldn't explaining these be the realm of technical people? Why should a manager care?

Two reasons. First, it's the largest cost. Second, all decisions, both the technical and the business ones, must be based on accurate information.

Put it into perspective: Your system is much larger than humans can read in a reasonable amount of time. A report about your system that is built manually will be at least inaccurate, but most likely wrong.

All decisions about your system must relate to the reality of that system. Everyone must care about how reliable and representative the information is.

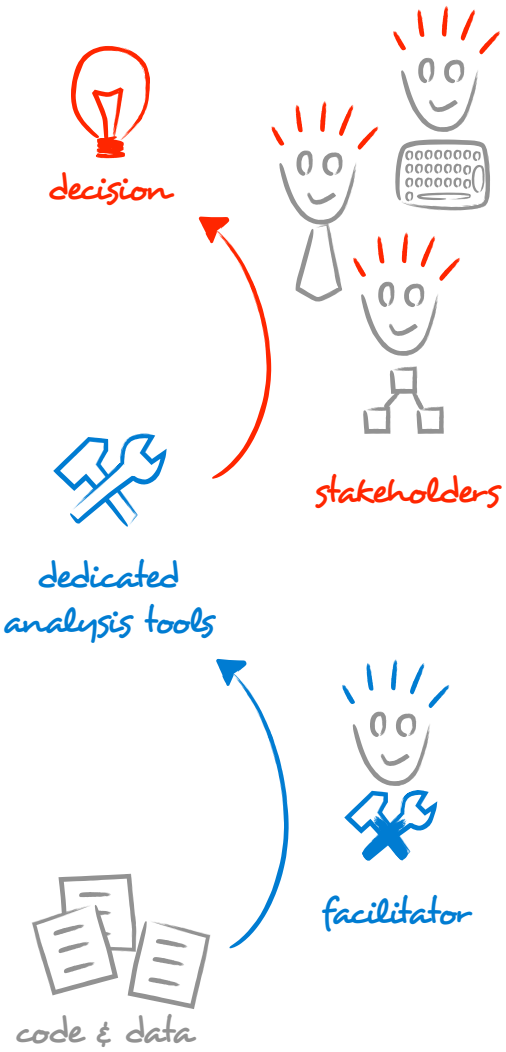
That's not only a technical issue. It's a business one, too.

Software assessment is a strategic skill.

For software systems to remain valuable, they have to be adapted to changes in the environment. The evolution challenge is posed by its internal structure. As the dependency on software increases and the need to change it becomes ever more critical, it is no longer enough to treat software as a black box: the ability to reason and decide about its internal structure is critical, and software assessment becomes a strategic skill.

This is relevant both when working with in-house systems, and when working with external providers. The assessment skill offers an infra-red like ability to identify and react to problems before they escalated.

It's like data science for software. Automating how information is gathered from the system reduces risks and frees energy that can be used for experimenting and acting.

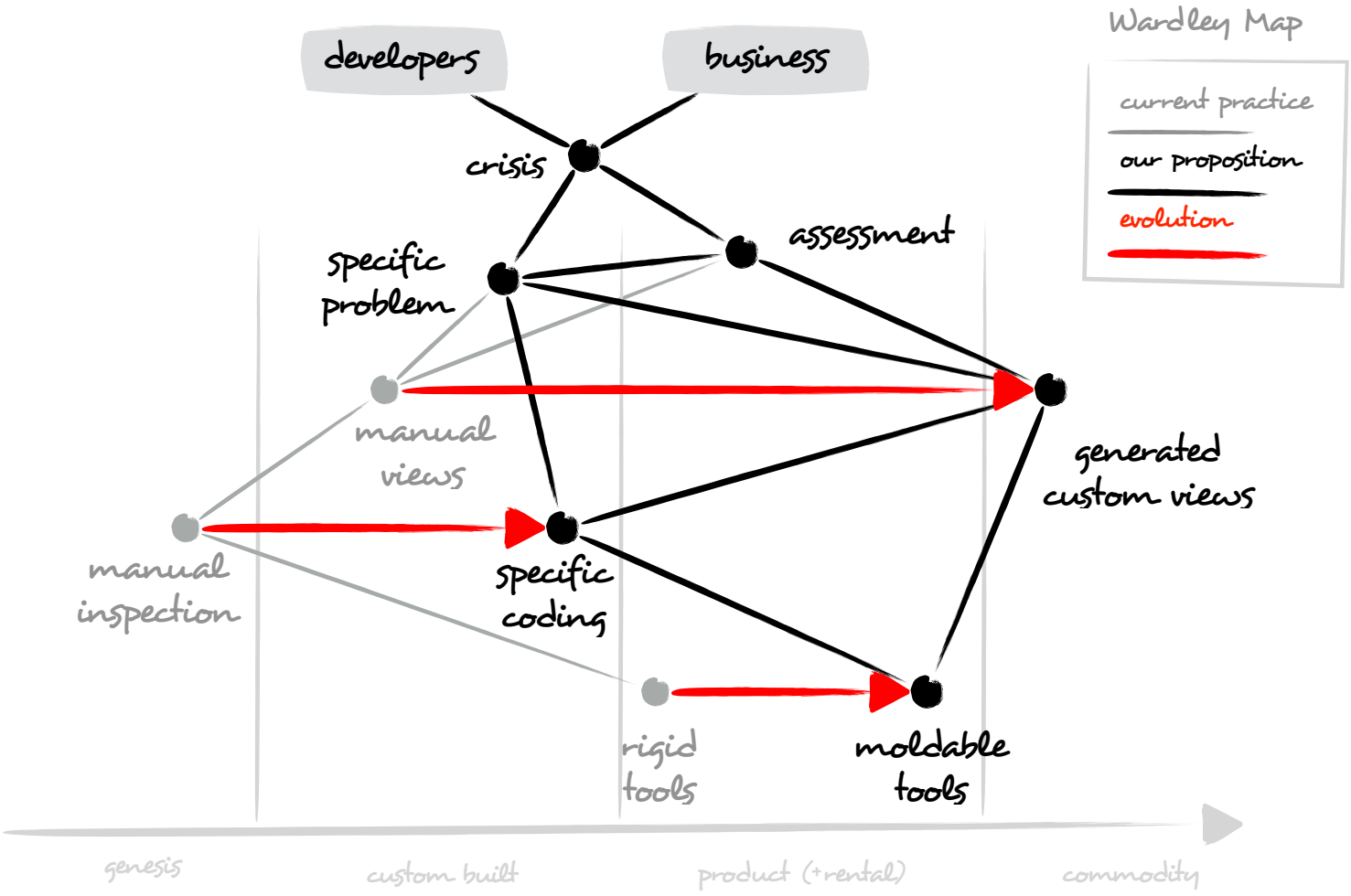


It's like data science, but for software.

Legacy is hard enough.
Eliminate the unnecessary risks.

The core proposition revolves around replacing manual views created through manual inspection by views that are generated automatically, yet specific to the problem.

This specificity is critical. Your system is special and so are the problems that appear in it. You want automation that serves that context because that is where value is. These views are created through specific coding that relies on new kind of tooling through the creation of these views becomes too cheap to matter.



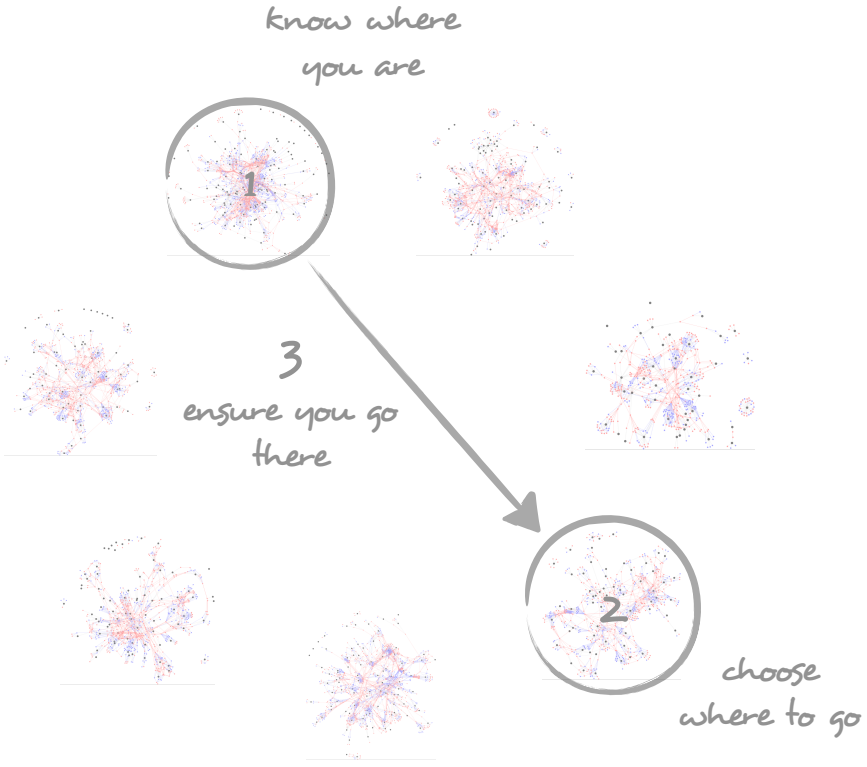
The ability to change the system tomorrow depends on its internal architecture. As the ability to change is of critical importance, it follows that the architecture becomes a business asset. And, like any business asset, you should treat it as an investment, too.

The only architecture that matters is the one that gets reflected in code. A key challenge is to steer the architecture while the code changes continuously. This implies at least three things:

1. know where you are,
2. choose where you want to go,
3. ensure you go there.

Point 2 is a design problem that is often covered well. 1 and 3 are assessment activities. Ensure you cover them well, too.

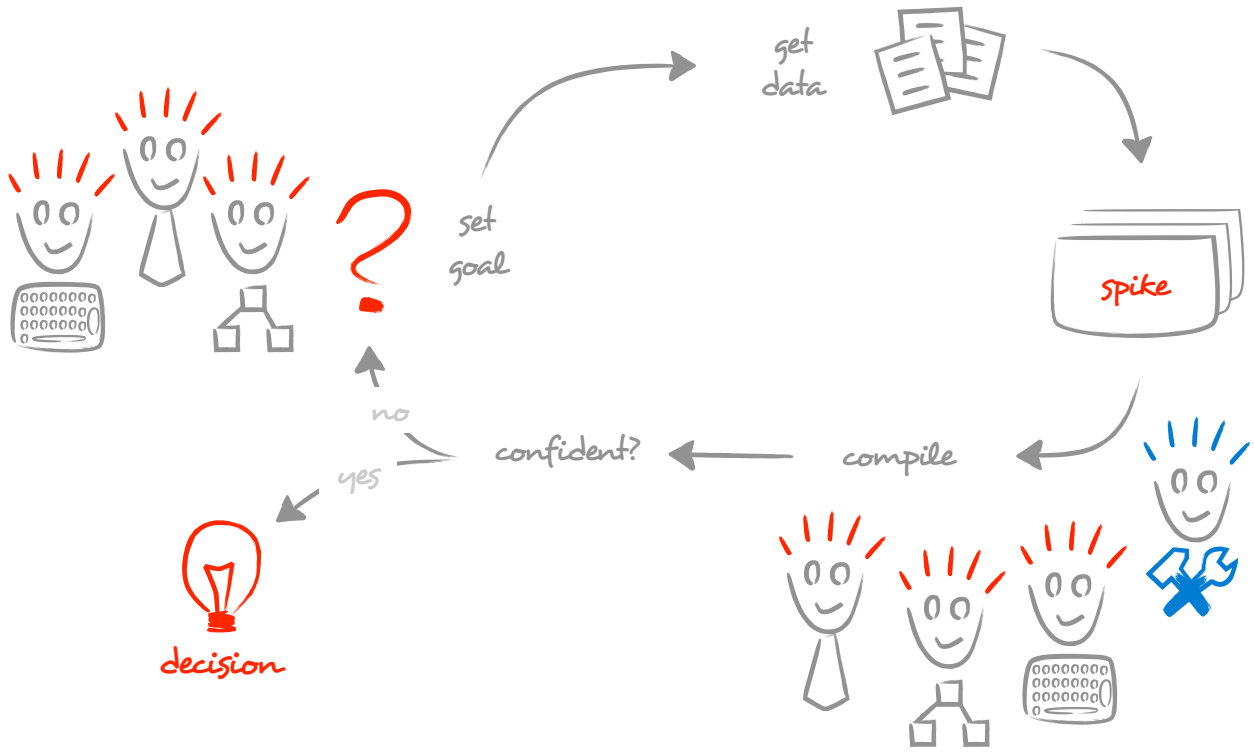
Architecture is a business asset. Steering it relies on assessment.



Explain the strategic problems that make a difference.

Often in crises, everything looks like an insurmountable problem. Of course, they are not. Relying on a systematic approach helps distill the relevant ones.

A crisis has both technical and business aspects and can only be addressed effectively through a tight collaboration between technical and business people. That, in turn, requires a common understanding and agreement on goals to carry on experiments and reach a concrete path to action.



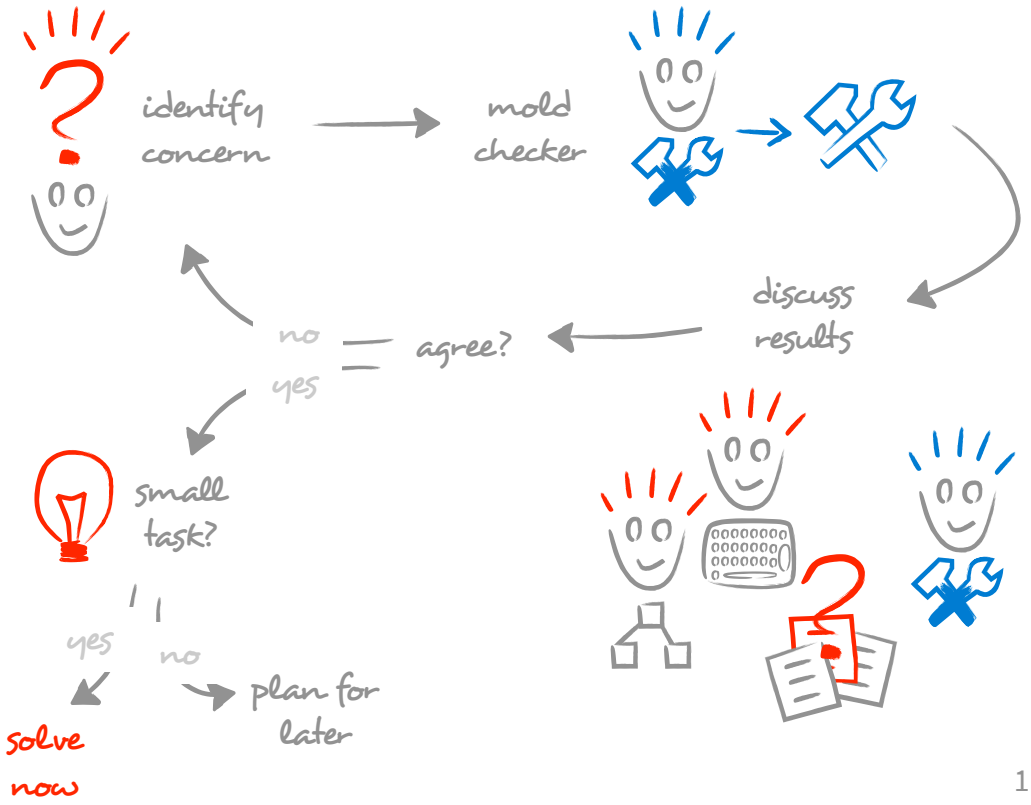
Explain your architecture explicitly. Steering it is a continuous investment.

The architecture that matters is the one that is eventually reflected in code. This means that those that affect the code are the main architects. You do not have one architect and many developers. There are only many architects. This means that architecture is a commons; a negotiation between different perspectives.

Design the organization to facilitate this negotiation explicitly.

Enter the daily assessment process. Anyone can raise a concern. A facilitator crafts the tool and the results are discussed in a common space or standup. In this standup only people that have data-based results can speak. This ensure crisp conversations. And at the end, the group distills concrete actions that are acted upon.

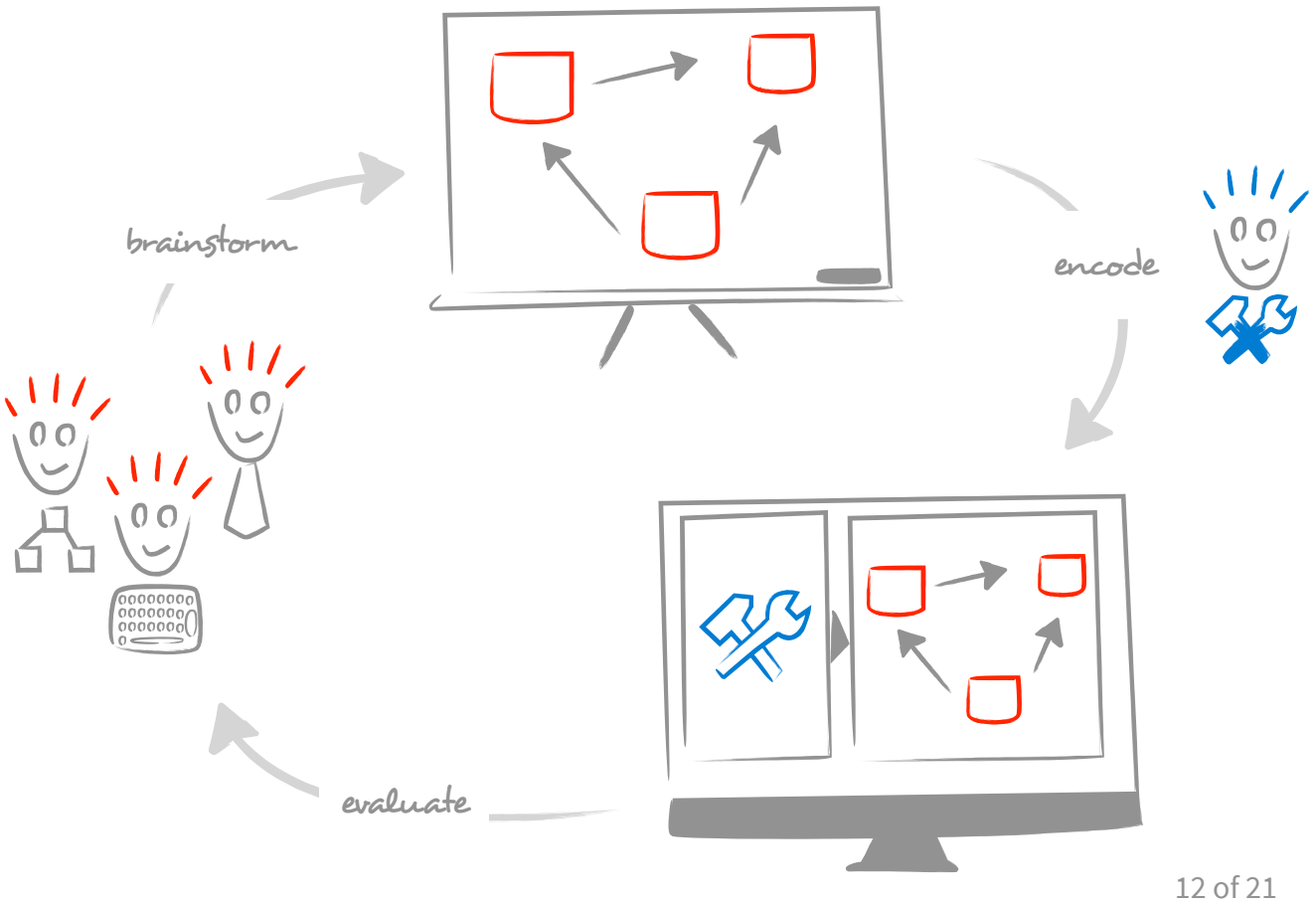
This seemingly simple process, enables the team to continuously identify, check and fix relevant technical concerns both about details and about broad architectural issues. This process provides the basis for steering long term migrations, while still



Explain your understanding of the domain in an executable form.

The domain knowledge is too often buried in implementation details. This makes reasoning about the system hard. Migrations, for example, often fail for this reason. A domain-driven ubiquitous language is important for bridging the gap between business and technology. This language is often captured during brainstorming sessions on whiteboards. But, that language must not remain only on the whiteboard.

It should be the responsibility of the system to provide the relevant pictures. This requires an explicit focus, but once in place, it changes how the business relates to the inside of software systems and it speeds up iterations dramatically.

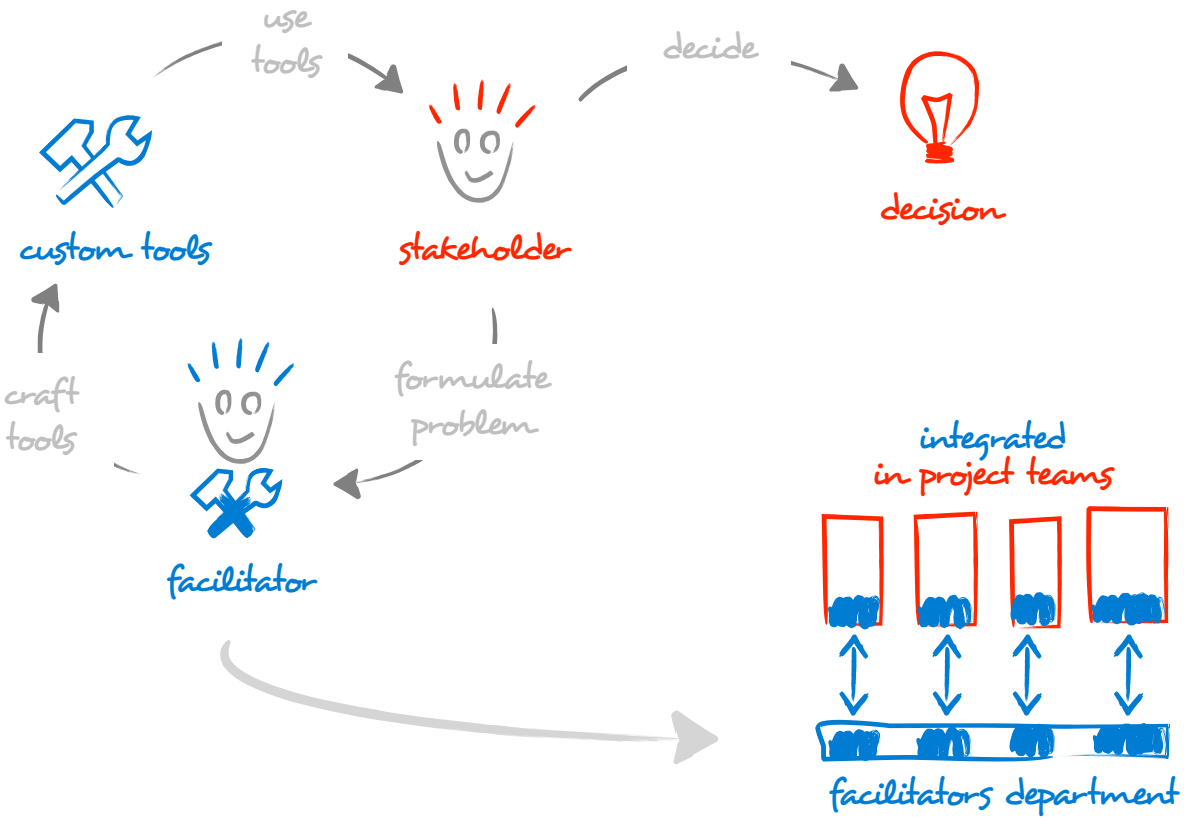


Explainability is not a recipe. It is a systematic discipline requiring dedicated skills.

Legacy means value, but value is always specific. There are no recipes to deal with it. It relates to your technology, to your domain, to your business. However, there are patterns you can learn and skills you can build.

To steer legacy effectively, you need custom tools that fit the problem.

Producing these tools requires dedicated skills. That's the job of the facilitator. But, the most important role is the stakeholder who should change the processes to integrate the new tools into the daily work.



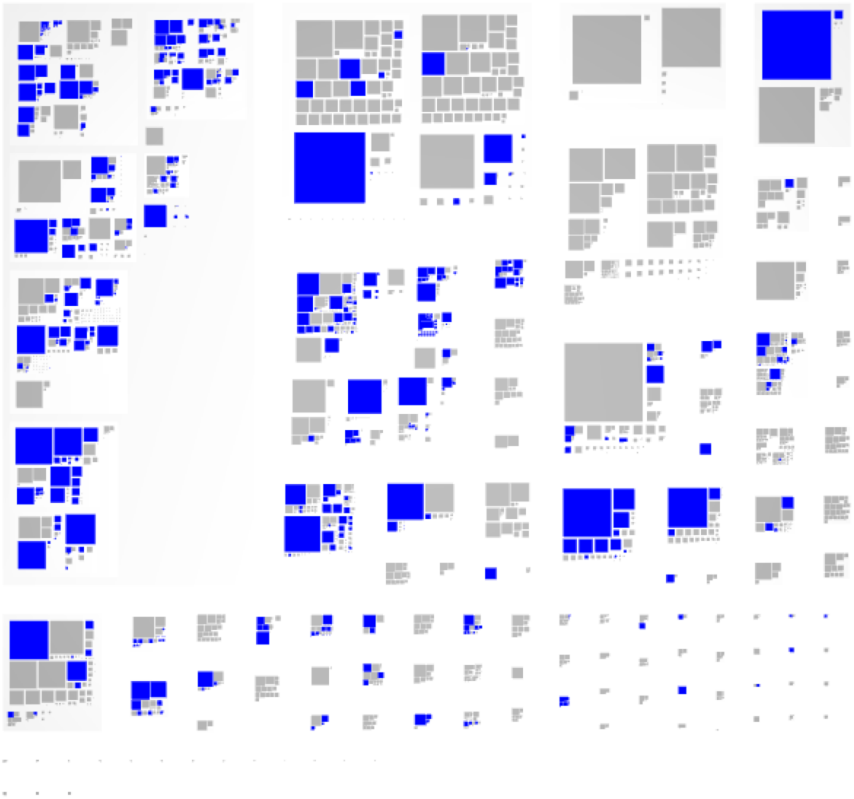
Scenario: Guiding a migration

Migrations are alluring. And they are risky.

The are alluring because of the promise of the new technology. They are risky because of the messy reality of the existing system. Yet, the success of a migration depends on the ability to assess that reality.

For example, in a migration case, the team manually estimated that a part that was to be migrated was only used in a few places. To validate this assumption, we created a custom analysis. The visualization below reveals (in blue) that, in fact, the part was used throughout the whole system.

Migrations are indeed risky, but the risks can be mitigated when they are visible.

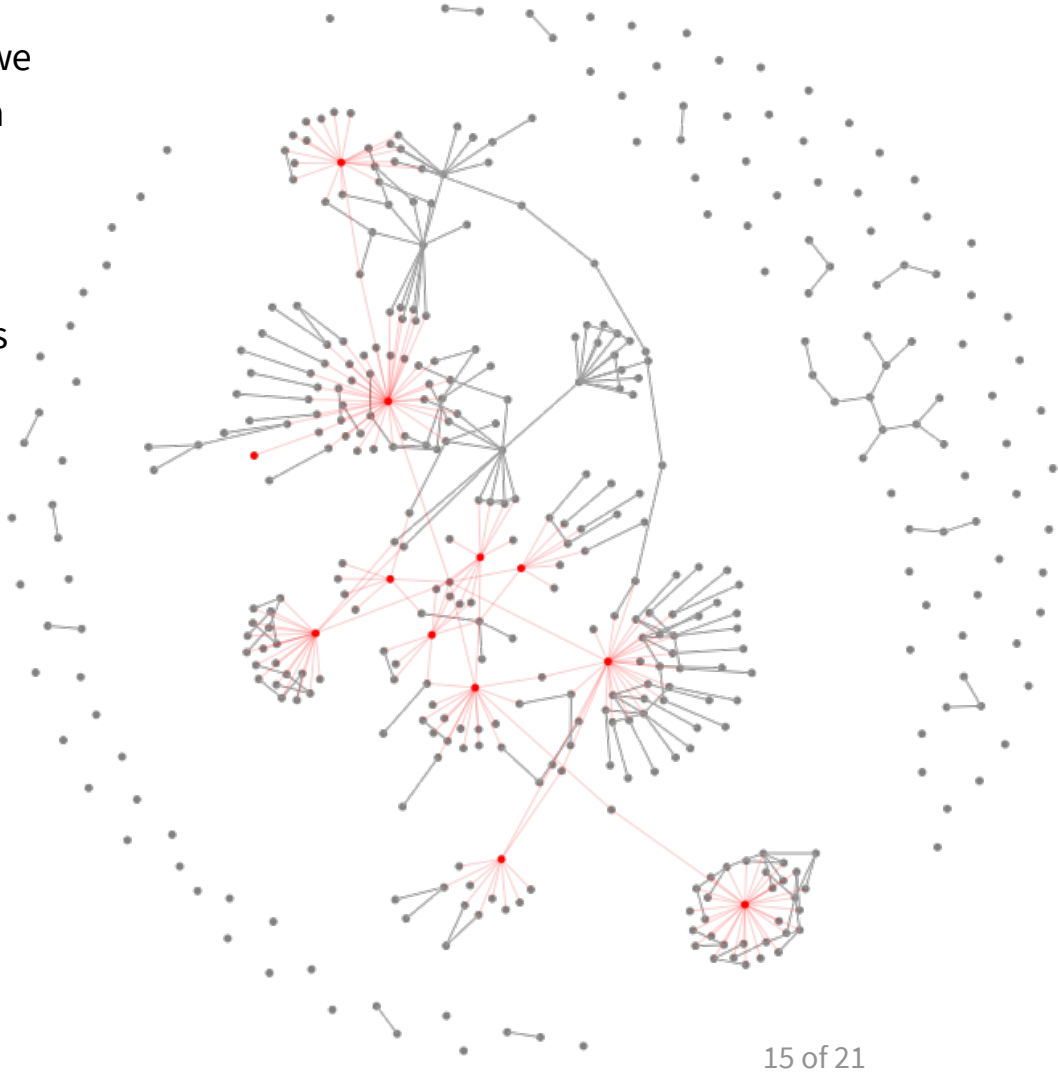


Scenario: Splitting a monolithic application

A monolith grows over time. It gets more and more important. It accumulates capabilities until it reaches a point at which, for various reasons, it does not scale anymore. That's when you need to split it. Yet, doing so is hard exactly because it grew organically without clear separations between its inner parts.

The splitting strategy must not only take into account the functional side of the system, but its internal structure, too. As that internal structure is usually specific to the system you need custom tools that help us think about unwanted dependencies.

For example, to the right we see a custom visualization showing the services of a system, in gray, and how they are used by their clients, in red. The clusters reveal splitting options.



feenk

We make your systems explainable.

We cover the whole legacy lifecycle.

From figuring the path forward, to steering migrations and to guiding rewrites.

Strategic assessment

We start by diving into your system. This is an intense, typically 4-8 weeks period to learn your context and distill a path forward. We work closely with you and guide the whole process through custom tools to explain the problem. The result is a concrete description of options and recommendations.

Steering migrations

Whether you migrate to a new technology, split the system into smaller pieces, or move it to the cloud, it is the existing architecture that poses the greatest technical challenge. We accompany the team through the process of steering the architecture in a new direction, and we coach the team to guide it by means of automatic views and constraints.

Guiding rewrites

Sometimes, building anew is the only reasonable choice. A new system requires discovery guided by a ubiquitous language that bridges the technical and business worlds. But, that language should not remain on a whiteboard. We make the system explain itself by automatically generated views. Through this we enable faster feedback and iterations.

	Strategic assessment	Steering migrations	Guiding rewrites
We identify technical problems by interpreting the business and technical context.	✓	✓	✓
We identify and discover domain concerns.	✓		✓
We assess systems and we architect transformations and migrations.	✓	✓	
We construct custom tools that technical and non-technical people use to make decisions.	✓	✓	✓
We document architecture through automated checks.		✓	
We document the business domain through executable views.			✓
We coach teams to incorporate our techniques and tools to guide the evolution of their systems.		✓	✓
We coach businesses to invest in explainable systems.		✓	✓

We are consultants.
We are researchers.
We are authors.

We bring a unique experience. We cover the whole spectrum, from a single line of code to decisions made at the company executive level.

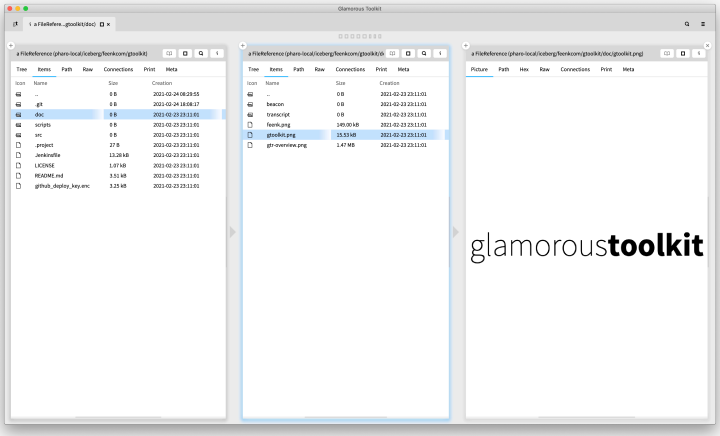
Our work is based on state-of-the-art scientific work, much of which we personally authored. We actively create new tools and techniques for thinking with and about software systems.

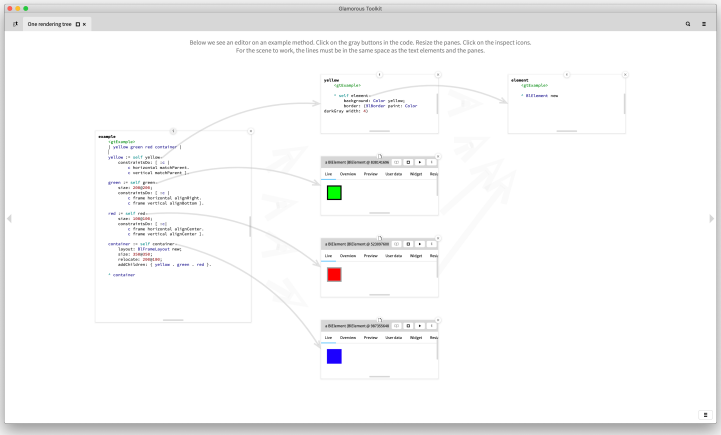
Our work has been validated for more than a decade of working with highly difficult problems in legacy systems.

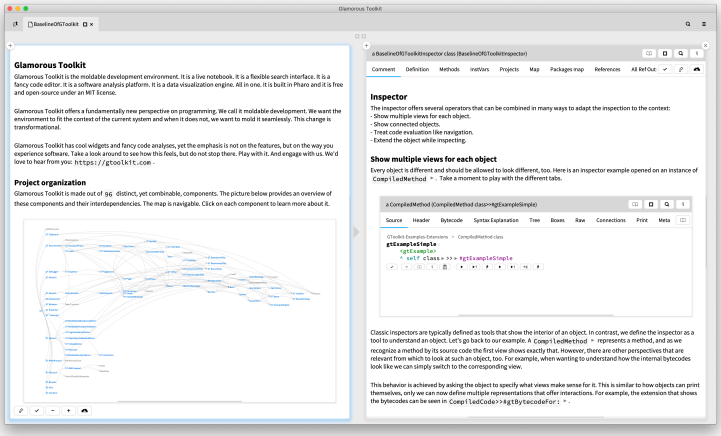
Glamorous Toolkit is the moldable environment.

Glamorous Toolkit is our highly integrated and moldable environment. It is a software analysis platform. A live notebook. A knowledge management platform. A rich visualization engine. A powerful query tool. A fancy editor.

But, most importantly, it can be molded in many ways to fit the context of the system at hand. This ability is crucial. Through it, decision making becomes both highly effective and a beautiful experience.







Before we part …

Software is not a cost.
It's an investment.

Black boxes are (too) risky.
Software assessment is a strategic skill.

Profitable systems are explainable.
Architecture is a business asset.

Hand-drawn pictures represent either wishes or beliefs.
Decisions should be based on facts.

Tools matter.
Pick them carefully.

Legacy is a positive thing.