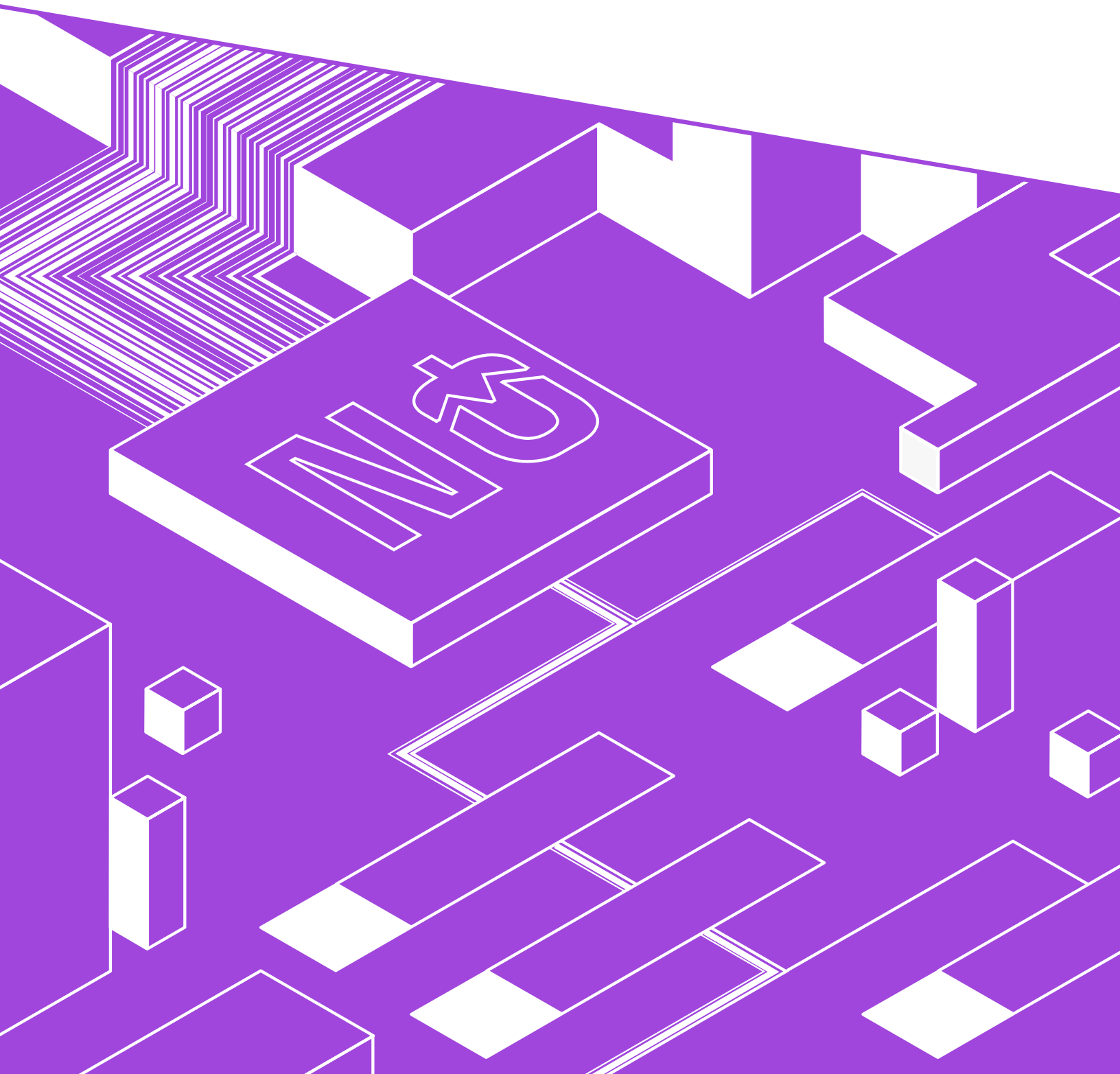
Nussknacker

Low-code for the business

Where it fits in your architecture



Introduction

The democratisation of application development is a natural next step for the IT industry. As the technology matures and its user base grows, more and more people are able to take advantage of it without (or with limited) help from technical experts.

Still, the flood of low-code tools may seem to be another example of companies trying to jump on the hot buzzword bandwagon. But that's actually not the case. It's a sound idea to enable business users to create or modify applications faster and at the same time relieve IT from mundane, repetitive tasks.

However, not every enterprise application should be developed by business users. And it's not only a matter of the current low-code tools' maturity. More sophisticated systems require internal or external integrations, there are performance considerations, security considerations, data integrity and so forth. These features are best designed and implemented by engineers, according to the specific requirements of a particular domain and the surrounding IT landscape.

It doesn't mean, though, that there is no place for low-code tools in more complicated enterprise applications. We believe there is, but that raises two questions. Firstly, which parts of a system are the best candidates to hand them over to business users? And secondly, what traits should a low-code tool have to help with that? In this white paper, we will try to answer those two questions.

Defining policies

The sweet spot for a low-code tool

The traditional or basic approach is that developers get (or gather) requirements from subject matter experts and implement all business logic in the code. The next step is usually to give business users some way to change some of that logic themselves. If done right, it's beneficial for both sides. But this traditional, all-in-the-code way has been given a lot of thought through all the years it's been used and can give us some insights into which parts of our applications we should aim to hand over to business users.

A well-organised code follows the principle of separation of concerns: different responsibilities are split between separate parts of the code with well-defined interfaces. How do you make that split? One of the most important rules is not to mix technical details with business logic. Implementation details – such as the choice of a messaging system, transactions, handling concurrency, security etc. – require technical skills and should be left to developers.

So the first step is to extract the code that, in one way or another, represents business logic. But developing these parts of the system may still require technical expertise, because business logic consists of many parts, behaving in different ways and having different requirements.

Let's take the domain of a bank as an example. The basic accounting rules (double entry, the sum of withdrawals must not exceed deposits etc.) are fundamentals that are critical to system correctness; they also change relatively rarely.

However, other rules, like how to calculate transaction fees, can change rapidly and they are based on current business ideas.

Policy - a design pattern that separates a varying algorithm from the process that it is used in and actions that it triggers.

In programming, a piece of code that encapsulates such variable decision rules is often called a 'policy' (it was originally defined in Domain Driven Design, a well-known methodology in computer systems design). The idea is that you have many algorithms – policies – that are implemented separately from the code that uses them and the system can choose at run time which one to use.

The original policy concept was used to describe a way of structuring code in a single application. However, with modern service and event driven architectures it's possible to move this concept to a higher level – where policies are defined and executed with a separate component, while other parts of the system are implemented as code.

Now, what can we learn from all of this, to help us find parts of our system that we could hand over to business users in a low-code tool?

The key observation is that not only the purely technical aspects of the system but also the fundamental, core logic should be designed and handled carefully. It requires not only good knowledge of technology but also proficiency in engineering practices. In such cases, it's the developers who should analyse business requirements and translate them into technology.

But with policies it's different: the translation from business requirements to code should be rather repetitive and straightforward (if the 'separation of concerns' was done right). Specifically, most changes in policies shouldn't require any changes in technical aspects, e.g. integration. We also don't have to be so careful – the main goal is to bring in the changes quickly and see the results. This means that, with the right tool, subject matter experts are able to instruct the system on what to do, without engineers' help.

For these reasons, we believe that in enterprise applications a policy is the sweet spot when it comes to how low-code tools oriented towards business users are used. Now it's time to answer the second question: what is 'the right tool' to manage policies?

A tool for defining policies

What should it look like?

In this section, we'll explore the qualities that should be considered when choosing a low-code tool to define and manage policies. We consider ease of change by non-technical users to be the main priority which should drive other requirements.

Ease of use

The main promise of every no/low-code tool is that it's easier to use than writing code in the standard way – i.e., with IDE, CI/CD etc. In a tool for non-technical users, it's even more important, but at the same time harder to achieve.

Guide the user

One of the most important facets of this promise is that the tool should guide the user. It can be achieved with various means – smart code completion, detailed and helpful validation messages, usage tips, templates and documentation and so on.

Make the user feel safe

Another important aspect is safety. One of the biggest barriers to empowering non-technical users to create logic in a system is the fear of introducing fatal bugs which break the system or its key invariants. This is especially true when it comes to a tool for defining policies. The only errors that should be possible are the 'business' ones – the user may incorrectly define the amount of a transaction fee, but the invariant that it is calculated and added only once should hold. For a user to feel safe, it should also be easy to test a policy before being used in production (see: Testability).

Separation of technical details from business logic

Non-technical users should not be bothered with implementation details. There should be a clear separation of roles when using a low-code tool to define policies.

Tool tailored to the domain

The configuration of the tool itself should be done by IT, handled with proper deployment/installation process and may sometimes involve coding. The result should be a tool that is tailored to the specific needs and domain.

No technical details exposed

In such a tool, end users are not required to specify technical details like URLs, credentials, timeouts, format of the data etc. They also don't have to understand and use mysterious terms such as 'Kafka', 'DynamoDB' or 'CLIENTS table in PostgreSQL', but rather familiar terms such as 'Sales events', 'Communication actions' or 'Client profile' etc.

Only then will the end users be able to efficiently implement the business scenarios of their particular domain. This is in contrast to, for example, low-code tools created for data integration, where technical details are often the most important thing.

Understandability

The way a policy is expressed in the tool has to be easily understandable not only by the users that create it but even more by those who will analyse and modify it.

Flow diagrams

A good example is decision tables – they usually take the form of a spreadsheet, which is well-known to almost all domain experts. Another example is diagrams similar to flow diagrams – they resemble Visio diagrams, which are again a standard way of showing business logic by business experts. By contrast, SQL is great for describing data joins, but it can quickly become hard to understand when conditions are complex and irregular.

Presentation fit for purpose

It is also important that different parts of an algorithm may require different presentations: a complex scenario may involve a decision table, invocations of ML models and simpler rules, all being part of a flow diagram.

Testability

For developers, testing is the basic tool that allows them to create correct applications – both automatic testing and the ability to easily run an application locally (on their own machine) to check how it behaves. In order to hand over managing policies to subject matter experts, it is necessary to provide them with a possibility to test the logic they define.

Test cases from real data

Sometimes the logic depends on a lot of additional data (e.g. client profile). In such cases, it's not convenient to manually pass all the test data. It may be much more efficient to be able to generate test cases based on 'real' data (of course, it may come from e.g. test accounts, if data security precludes using production data).

Painless testing

Besides helping with verification of a policy before it's deployed to production, testability should also support experimentation. Testing must be simple and quick enough to let users see partial, intermediate results of their work, and to safely explore different bits and pieces of the system.

Observability

In software systems, observability has become one of the hottest topics lately. It is a powerful idea that we should not only be able to monitor the state of a system, but also to track how individual events or user interactions were handled.

Monitoring business metrics

Non-technical users have the same need – to be able to understand how their policies behave – but their tools need to be focused not on technical metrics, but more on business ones. It can be simple, like how many events failed this condition, or more complex, involving tracking a single business event throughout the whole system.

The context of policy makes observability even more necessary. We want to make changing business logic easy and cheap to allow users to modify it as often as required. But that means they have to be able to constantly monitor the results.

Logic 'language' easy to monitor

The design of a low-code tool for policies should also consider the fact that various forms of describing a policy differ in how convenient they are to monitor. For example, a decision tree makes it easier to show how each condition performs in comparison to a complex SQL query.

Governance

In a large organisation, there may be dozens of sources of data and hundreds of different business policies. The ability to track changes and audit their authors is obvious.

Visible impact of changes

It is also vital to be able to easily understand where a specific event or piece of data is used, for instance, to assess the impact of a change, or if it's possible to remove the data from the source system, the concept known as data lineage. Sometimes it's also necessary to track usage of certain sensitive data fields.

Performance

Real-time ready

Real-time decisioning became a necessity for many organisations. Real-time marketing, handling fraud detection and reacting to signals from sensor networks are just a few examples.

In all of those use cases, business logic can be encapsulated in a policy. If we want to use a low-code tool for that, it has to be able to handle large amounts of traffic and apply the policies efficiently.

Independence from a processing paradigm

The idea of externalising policies to a low-code tool entails a change in architecture: other components of the application have to be integrated with the tool. However, this shouldn't limit other architectural choices.

All data processing paradigms

An ideal policy-management low-code tool should fit different data processing paradigms. Whether it's asynchronous processing of data streams, a synchronous request-response model or even batch processing, it should be as straightforward as possible to integrate for the engineers and mostly transparent for business users.

Nussknacker as a tool for defining policies

We strongly believe that externalising policies to a low-code tool gives both business users and engineers the most benefits. Non-technical (or less-technical) business users get the ease of changing the behaviour of the system in aspects that really matter to them. IT departments get rid of mundane work without the fear of losing control of the aspects of the system that are important to them.

Nussknacker is our take on the idea. We are creating it using the qualities defined above as guidelines:

We decided to use flow diagrams to make it easier for non-technical users to create and analyse scenarios. This makes it natural for users used to describing logic with, for instance, MS Visio diagrams.

Simple yet expressive expression language is used to define various conditions and rules – it has extensive validation and smart, context code completion capabilities. It's also easy to add new functions suitable for a particular domain.

The integration with Kafka, OpenAPI or ML models repository is done as part of system configuration, not as a part of scenario authoring. For example, when creating a scenario based on Kafka events, the user only selects the topic, but not the Kafka connection details.

The components used in the diagrams can be configured (names, defaults, links to documentation) to better align to the particular domain.

Scenarios can be tested before deploying to production. Testing is embedded in the tool so it doesn't require any additional scripts or environments. Input data is provided by the user or can be generated by Nussknacker.

Enrichments in tests can be done against real services (if there are no side effects) or can be run on mocked ones.

Nussknacker has been created with performance in mind from the very beginning – some of our first deployments are processing >150k Kafka events per second, running on Apache Flink. It's also possible to use Nussknacker in request-response architecture – each scenario can be viewed as a REST endpoint.

If that sounds interesting and you would like to learn more, we encourage you to visit **nussknacker.io** or reach out to us.