Designing with Intent: Layer Two Philosophy

1. Introduction

1.1 The hidden complexity of clever solutions

During the final months of my time at university, I was introduced to Robotic Process Automation (RPA) during my internship. It was the first time I realized that in IT, theoretical solutions don't always translate well into practice. In this case, we were trying to make sense of the infamous Excel macro jungle and trying to turn it into low-code automations. What started as a clever way to automate repetitive tasks quickly turned into a tangled mess of hidden logic, undocumented dependencies and fragile workflows. The moment someone asked, “Can we make this into something more future-proof?” the answer was usually followed by numerous questions and visible frustration.

That situation is not limited to the Excel example. It's a pattern we can observe in any organization, using any tech stack. When solutions grow organically, usually based on enthusiastic early adopters without clear boundaries or long-term vision, they become harder to maintain, harder to scale, and harder to explain.

1.2 Problem Statement

At the company I currently work for we are starting to see signs of this pattern too. The solutions work very well, they add a lot of value, but it's becoming increasingly difficult to keep them agile. Introducing new features often requires changes across multiple layers. Fixing issues can take multiple sprints and responsibilities between the different teams are not always clear. In short, it slows down development, increases the risk of new problems being introduced, and makes onboarding new colleagues more challenging than it should be.

That is why we came up with 'Layer 2 philosophy'. It's more than just a component or part in technical designs; it is a structured way of thinking about the way in which to decouple/modularize systems to enable more flexibility and maintainability. Not just for technical elegance, but to create a layer of abstraction that helps separate technological challenges, define clear boundaries, and build with future change in mind.

1.3 Motivation

We are at a point where we need to conciously make design choices. Both our platform and ambitions scale and we need architecture and solution designs that support that growth, preferably without any major impact. Introducing the philosophy in team discussions or with management has proven challenging without a structured explanation. It needs a more professional approach. It needs a clear *why*. And that is why we write this essay.

This document is a starting point for that discussion. It's not a blueprint, but a philosophy. Alignment of core principles won't just create better solutions, we'll build a more agile way of working. In the sections that follow, we'll outline the scope, principles, design choices, and take a look at practical implementations.

2. Scope

2.1 Feature Reference

The Layer 2 Philosophy has been in the back of my mind for a long time. But until recently there was never a clear reason, or the right moment, to write it down properly. That changed after a recent brainstorm where we addressed the complexity issue. The idea to work out a strategy for this was added to our backlog, although without a clear scope and list of deliverables.

Through this essay, the goal is to give that feature what it's missing: a scope, a set of guiding principles and suggestions for both improvement and critique.

2.2 Definition

Right away, it's important to realise that Layer 2 is not a silver bullet. It's not a new platform or a strict set of rules. It's not a tool or a framework. It also does not mean a complete rewrite of existing solutions. Instead, it's a way of thinking about how we modularize our solutions. It sits somewhere between architecture and design choices, both technical and conceptual.

A more technical explanation: It allows two subsystems to evolve independently without requiring a full redesign of the entire solution. As an example: UiPath Orchestrator (here the data providing subsystem) tracks user actions in the form of audit logs. OpenSearch (in this case the data consuming subsystem) allows us to create alerts based on rules it can find in its data. If we were to directly connect these two systems and we change either of them, a complete rebuilding of the entire solution is needed. Building a layer in between the two subsystems allows not only the individual transformation of the two subsystems, but also for us to keep our calculations or data transformations to stay intact and easily maintainable in one place.

3. Principles

3.1 Core Tenets

To make sure we are on the same page, let's start by explaining the word *tenet*. A tenet is not a rule, it's not a fixed truth or fact. Instead, it is a belief. A tenet is something we hold to be valuable and that can change over time. In the context of our philosophy, tenets are living ideas. They are open to discussion, require refinement and even contradiction. It is important to remind ourselves of this, because the solutions we build are never static, they're also bound to change. In summary, if our systems grow and change over time, the ideas about how to build them should be just as flexible.

Back to our three challenges:

1. Unclear responsibilities.
2. High effort to maintain and respond to planned changes.
3. Low agility when dealing with unplanned changes or future needs.

The three problems will be addressed by following the *logical progression* below.

3.1.1 Modularize

The first step is to think of the current solution architecture and break it down into smaller components, each with a specific purpose. We call this modularization. But it's not just about splitting things up, but doing so with intent. We should ask ourselves: "How far should it go?" The answer is somewhere in between, far enough that each module can be understood, tested and maintained independently, but not so far that modules become fragmented or unnecessarily complex.

Modularizing a solution allows the team to isolate responsibilities and makes it easier to assign ownership of a part of a solution to a specific group or person. Later in this essay, we'll also look into the downsides, because there are also trade-offs to be made.

3.1.2 Abstraction Layer

Once the solution has been modularized to whatever extend fits best, a new layer is required to connect the components. This is where a new abstraction layer comes in. The layer, like the previous example of UiPath Orchestrator and OpenSearch, and *translates*, *filters* or sometimes *restricts* the interactions between the two systems.

3.1.3 Decentralize

Now we have multiple components, with an abstraction layer between them. In other words, the solution has been decentralized. Teams or individuals can own a module or layer and make changes without depending on others. This decentralized outcome is what helps us solve the three challenges.

Challenge 1: Unclear responsibilities

We now have a separation of duties, adhering to the so-called *single responsibility principle*. It will lead to better documentation, easier onboarding of new employees, more explainable systems and simplified interactions and dependencies between teams.

Challenge 2: High Maintenance Effort

Bugs or other issues can be fixed in an isolated part of a solution without touching other layers. This reduces things like overhead, but most importantly it simplifies change, planned or unplanned.

Challenge 3: Low Agility

Solutions will be more future-ready, or can easily be made more future proof, without breaking the entire system. Whether it's technical changes or functional changes, there is now more agility.

The above is not just theoretical, it's also observable. We've seen them in previous developments. In the next section we'll look into them in more detail, hopefully verifying if the philosophy makes sense or not. But first, let's take a step back and ask: "How do these tenets influence the actual design choices we make, architecturally?

3.2 Design Philosophy

Answering that question can be done using the same three *tenets* as before; modularization, abstraction and decentralization.

Modularization

When we modularize, the goal is to make conscious decisions to separate logic, define ownership, and create isolated components. It shifts the design approach from building features to building with an almost LEGO brick mindset. The pieces have to fit other pieces easily, and more importantly without having to take apart the entire build.

Abstraction Layer

Whether the abstraction layer is an entire new solution or just an approach to linking two subsystems, the important part is about dataflow. What data (in- and output) flows between components, how is it transformed, and what logic belongs where?

This reduces the effort it takes to decouple solutions and makes it easier to swap out parts of the system without breaking everything else. It's a design choice that prioritizes flexibility over convenience.

Decentralization

Decentralization influences design mainly on the *who* part of the solution. It's where design strives towards autonomy and accountabilities. Teams should be able to own their modules and make changes independently, which means there should be restrictions on access.

3.3 Making it visual

These principles aren't just theoretical. We are able to see them in many modern architectural patterns such as microservices, event-driven systems or developer design standards. Over the years, the industry has already naturally moved toward a more modular, loosely coupled architecture for a reason. In the next section, it is shown that the philosophy works in practice by stating examples, and we look into which parts of our platform are candidates to verify our philosophy.

4. Common Candidates

4.1 Examples

We've described multiple principles or choices for the design of solutions, but none of them are new. They have been around for years and have proven their value, they might just be described in a different way than we did so far. The idea is to give the philosophy context, some guidelines, that assist us with the challenges. It's not to reinvent the wheel.

To help describe this, let's start by looking at the opposite: Monolithic architecture. It's how software used to be designed and is sometimes still used for smaller projects or in early-stage startups. It is still modular code and despite common misconceptions, it is not a bad way of creating a solution. It is often more simple, it's easy to deploy and cost-effective. However, when systems grow, the maintenance and flexibility of the solution tend to become harder. That is where a more modular approach proves to be better.

In the paper Software Architecture Evolution: Patterns, Trends, and Best Practices by Nivedhaa N., multiple principles of *proper architecture design* are outlined. They align closely to what we call Layer 2 and they're explained below.

Modularity

Defined as breaking systems into multiple components that handle specific topics, actions or concerns. It makes a solution reusable and simplifies maintenance.

Layered Architecture

This is when someone organizes systems into layers (e.g. presentation, business logic, data) to improve separation of concerns (in turn making modularization easier) and scalability.

Loose Coupling

Used to minimize dependencies between components to reduce the impact of changes.

High Cohesion

It might sound like the opposite of loose coupling, but this principle is meant to help create individual components and make it clearer what belongs to a single *module*.

Scalability & Fault Tolerance

This principle strives to design systems in a way so that they can handle growth and failure.

In addition to these principles, the paper also highlights design *patterns* that support flexibility and maintainability. They are called the Adapter, Strategy, and Observer patterns. These patterns help systems evolve without breaking, and they often live in the very abstraction layers we are advocating for.

In the next section, we'll look at two examples from our own organization that show how the principles from the paper play out in practice and why actively keeping them in mind during development is so important.

4.2 Use Cases

Example 1: WebDAV in Kubernetes

A few years back, when we still had a more simplified version of our Kubernetes (K8s) setup, we used WebDAV for file storage. It can be easily configured as a K8s object by specifying some YAML file and configuring it accordingly. Luckily for the team, there is a built in a abstraction layer already present, provided by Kubernetes. Without diving into too many details, here's a quick explanation of how it works.

Kubernetes works with so-called objects, they are *records of intent*, meaning the system will constantly work to make sure that the object you specify exists. In the case of the team's setup, there are multiple relevant objects: the webdav *pod* (basically the actual service), a PersistentVolume object (as the name suggests, a piece of storage that does not go away on restart/redeploy) and finally a PersistentVolumeClaim (that let's the volume know it will actually use that part of the storage). These objects are all individually specified and configured, they're the same layers of abstraction we introduced in the layer 2 philosophy.

The result of this setup? When we moved from Disk Storage to Azure FileStorage, and later to Blob Storage, the changes were minimal. The only change that was needed was the PersistentVolume object, and a few parameter changes in other objects. The abstraction allowed the team to swap without touching the core logic. It was modular and maintainable, and it's exactly what Layer 2 thinking is about.

Example 2: Hybrid Test Automation Framework

Another example comes from a test automation setup one of my colleagues used in the past. They made use of something called *hybrid test automation framework*. It's commonly seen in tools such as UFT, HCL OneTest (both enterprise testing software tools) or Selenium (opensource tooling, supporting Java, Python, C#, etc.). At a high level, the framework is represented by a single flow in the test tools, but underneath it's built from separate modules. For example, a group of logging actions, a data layer, and a group of actions that interact with the UI. Each of these can be developed, tested, and maintained independently. For instance, if a button moves or a selector breaks, we only need to update the UI interaction module. The logging and data layers stay (almost) untouched.

5. Current System Architecture

5.1 Current Architecture

The team I currently work in at Rabobank is responsible for multiple platforms that consist of a variety of components. For simplicity, we've grouped them into categories. While some categories, like the actual RPA workflows, are out of scope for this essay, the other solutions offer the opportunity to apply Layer 2 thinking.

Here's a simplified overview of the main categories:

RPA Workflows

The UiPath and Kofax automations we (or the users of our platforms) build and maintain. They're considered out of scope for this essay.

Monitoring

Everything related to creating, labeling, analyzing, reporting (dashboards), and improving the logs of our solutions.

Alerting

All solutions that inform the team or external teams about events, usually based on the logs from Monitoring.

ETL / Data

The gathering, movement, preparation, or calculation of solution data. Logs are excluded here and considered part of Monitoring.

Infrastructure- and Configuration scripts

Everything related to infrastructure components like Windows servers, databases, or cloud resources (like K8s).

Deployments and Retirements

Scripts and pipelines that promote processes from the first category to higher environments or retire them.

Shared Utility

Scripts or repositories that span multiple solutions or environments and serve as a basis for broader ecosystems.

Support Utility

Scripts or automations that support day-to-day work of platform engineers, such as elevating rights or managing access.

5.2 Approach

Looking at these categories, there are several areas where Layer 2 thinking could be introduced or expanded. But how do we start to identify those opportunities? We believe there is no need for a big bang transformation. Instead, we recently started to refactor multiple solutions, introducing one abstraction layer, and iterating based on feedback of the team.

Layer 2 should always serve a clear purpose. Whether it's improving scalability, reducing maintenance effort, or speeding up delivery, every architectural change should be driven by a clear goal. For example, modularizing the Alerting systems could help us reduce noise in incident responses.

Finally, we should treat this transformation as an ongoing process: We experiment, learn and adapt. Not every abstraction will work perfectly, and not every module will be clean from day one.

In short, the best approach would be to look at the solutions we are currently refactoring and see if they benefit from layer 2 thinking. In a few months we should evaluate and adapt, using questions such as:

- Are our systems more stable?
- Are deployments faster?
- Are bugs easier to fix?

6. Counterarguments and Critique

6.1 Challenges

In any philosophy there should be room for questions, discussions and answers, Layer 2 is no exception. We strongly believe it offers a structured way to improve, there are also valid concerns, or even risks, that need to be discussed.

More Components = More Maintenance

A common point of critique of modular architectures is that they introduce more moving parts instead of simplifying the solution. More components mean more configurations, more complexity and potentially more bugs that need solving. A solution that looks simple might end up with a complex web of dependencies that's harder to manage than the original.

It does indeed sound counterintuitive. Some people might argue that adding complexity just to make things *cleaner* is the exact opposite of what the new philosphy tries to achieve. And that though is not unfounded, Layer 2 does add extra layers.

How far do we go?

Recently, reading a paper called 'Software Architecture: The Hard Parts' helped me realize that this question is not about whether to split up parts of a solution, but it's about granularity. In other words, how small should the parts be when we split up?

Granularity isn't defined by the number of classes or lines of code. It is shaped by the people building the system (the developers) and the purpose each part serves. Using specific *drivers*, we can evaluate whether to integrate (merge) or disintegrate (split) components or services.

Integrator Drivers	**Reason**
Database transactions	If splitting compromises the integrity of the data, we should reconsider.
Workflow	If the workflow is complex and very cohesive, integration improves fault tolerance and reliability.
Shared code	If shared code or dependencies cannot be easily reused, merging components improves maintainability.

Disintegrator Drivers	Reason
Service scope	If the separate parts are not cohesive, they should follow something called the *single responsibility principle*. This means every part only has single-purpose actions
Code volatility	If the code of components changes at different rates, separating them reduces the amount of unnecessary changes.
Code scale	If components are very different in size or transaction scale, splitting them will prevent over-engineering the simpler components.
Fault tolerance	If fault tolerance requirements vary, separating them prevents unnecessary (often complex) robustness in simple components.

So, how far do we go? The answer is not fixed. It depends on the context, the trade-offs, and the goal(s) of the solution. But using a balance between the drivers below and discussing them in a team will assist in making that decision.

7. Conclusion

Layer 2 philosophy is not an instant fix, it is a change in approach. It's goal is to encourage a team to think beyond immediate technical fixes and instead design with agilit, clarity, and maintainability in mind. By consciously discussing to modularize systems, introducing abstraction layers, we create solutions that are easier to scale, easier to maintain, and more resilient to (future) change.

The philosophy does not come without trade-offs. It can at times introduce complexity and demands a time investment from multiple teams. But if applied properly, it can help us build solutions that evolve easily instead of after a lot of effort.