THE KLARRIO VISION
FOR ARCHITECTURE

MOVE FAST AND BUILD THINGS

Klarrio
STREAMING AHEAD

TABLE OF CONTENTS

01 INTRODUCTION. 02

02 INSIGHTS. 03

CHANGE IS THE ONLY CONSTANT. 03

YOU CANNOT RETROFIT SECURITY. 05

ELEGANCE DOES NOT WIN YOU PRIZES. VALUE DOES. 06

PROTOTYPES ARE A CONVERSATION PIECE. 07

TO MOVE FAST, YOU NEED CONFIDENCE. 08

CLOUD-NATIVE ISN'T JUST A BUZZWORD. IT'S A PHILOSOPHY OF DESIGN. 09

DATA HAS GRAVITY. 10

GOOD FENCES MAKE GOOD NEIGHBORS. 11

03 PRINCIPLES. 12

PROTOTYPE EARLY. PROTOTYPE OFTEN. 12

Prototyping as a Practice. 12

The Lasting Value of a Prototype. 13

Prototype to Accelerate Architecture. 13

CONTROL THE COST OF LEAVING. 14

CONTROL THE COST OF STAYING. 15

DREAM LONG-TERM. PLAN SHORT-TERM. 16

ARCHITECTS ARE ENABLERS, NOT BOTTLENECKS. 17

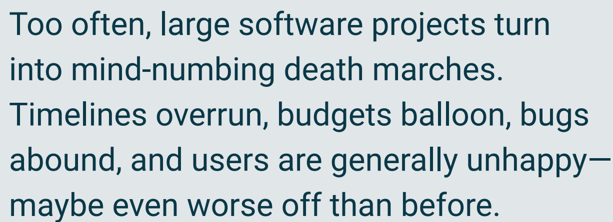
BE BRIEF TO BE RELEVANT. 18

SECURITY IS FOUNDATIONAL. 19

04 CONCLUSION. 20

05 ANNEX. 21

01



Too often, large software projects turn into mind-numbing death marches. Timelines overrun, budgets balloon, bugs abound, and users are generally unhappy—maybe even worse off than before.

An overly rigid approach to architecture is one of the biggest causes for each of these problems. In short: the belief that software should be complete and right the first time puts a stranglehold on the very software development process itself.

It's easy to see why strangleholds keep happening. For many engineering functions, instant perfection is the only viable mindset. Bridge builders don't expect their first three attempts at building to collapse, nor do they say: "This one's too narrow. We'll release a new version in three months that will support double the traffic."

But even in traditional, staid engineering branches, this conventional wisdom has been challenged. If you compare the budgets, timelines and achievements of NASA's Space Launch System to those of the more flexible SpaceX mindset, it's clear that even in Rocket Engineering, iterative approaches and the willingness to fail and improve in production bring value.

Yes, there are life-critical pieces of software that must be designed the NASA way. But let's not kid ourselves: 99% of software out there doesn't need to achieve such high standards. For this vast majority of software projects, a more nimble, iterative approach to software development can bring value and save projects from premature obsolescence.

With a nimble approach, software architecture plays a crucial, make-or-break role. Architects have the power to either choke a project to death by creating ivory tower designs and holding up progress while they work on the "perfect" solution, or they can assume the role of facilitators and supporters, helping developers build extensible solutions. These can both evolve with the times as well as deliver value early.

To cut right to the chase—we favor the latter kind of architecture.

Over the course of a couple of decades of development and architecture experience, we have gathered quite some insights. They have shaped the way we approach architecture practice today, and the goal of this paper is to share the most significant ones with you.

INSIGHTS.

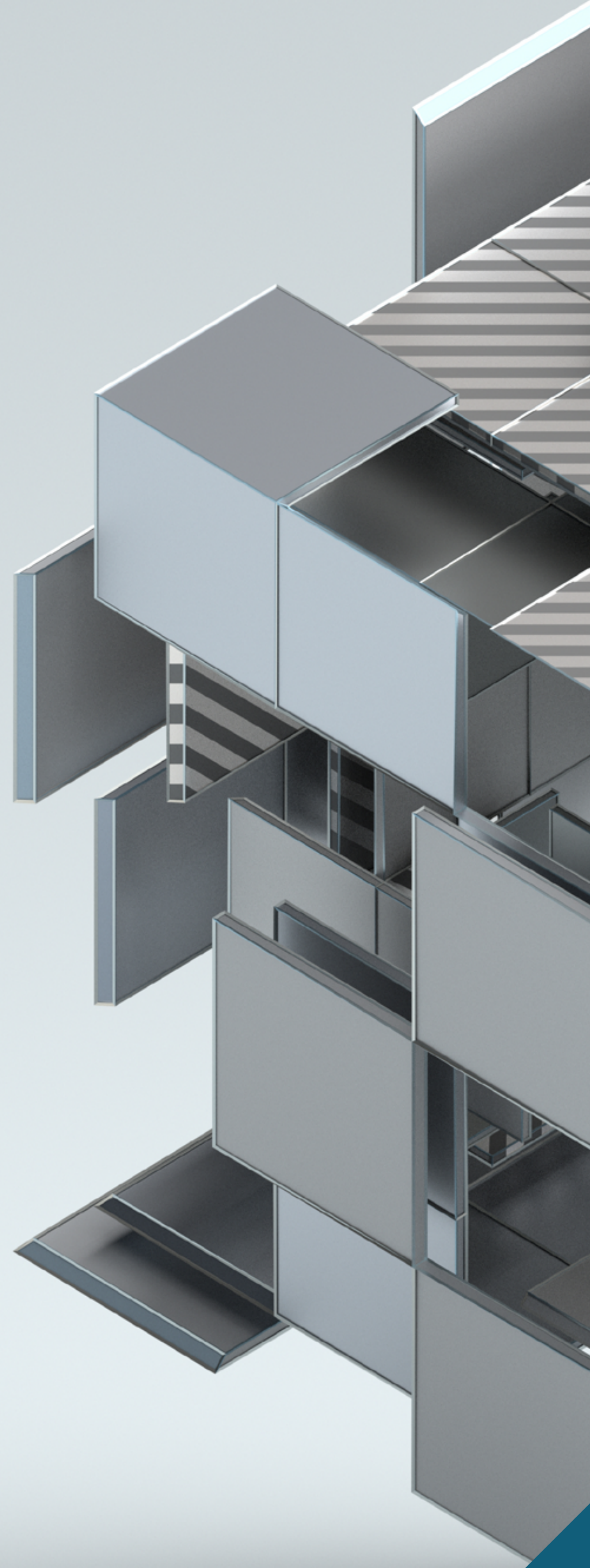
02

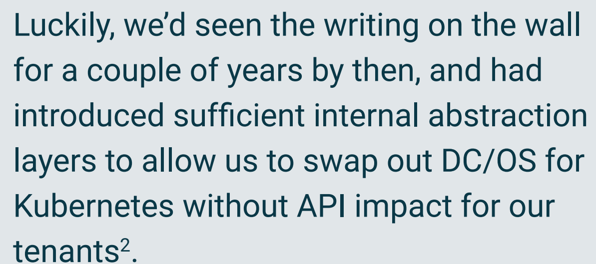
CHANGE IS THE ONLY CONSTANT.

Klarrio typically has multi-year engagements with customers, building secure large-scale data processing platforms. No matter how well laid-out the plans for such projects are at the beginning, we have never seen such a plan survive the first two years of engagement.

While you're happily beavering away building a platform, the world doesn't stand still. The context in which you're building your software is ever-changing. Business goals and priorities change. For example, regulatory changes like NIS2, CRA and GDPR force important new requirements, and sometimes a paradigm shift sweeps away all certainties you had at the beginning. (AI, anyone?).

Even on a technical level, you're constantly on the move. Today's increasing focus on security forces you to constantly monitor your project for (exploitable) vulnerabilities. More than ever before, you have to keep up with the latest-and-greatest versions of your building blocks to counter the constant barrage of security vulnerabilities out there. The rapid pace of innovation may also obsolete the technology choices you originally made, leaving once-promising technologies to wither on the vine.





Case in point: back in 2017, we designed and built a streaming data platform for smart traffic in the Netherlands[1]. We had decided containerized deployments were the future (it may surprise you, but back then that wasn't the no-brainer it is today).

In our search for a suitable container orchestration platform, there were two main contenders. On the one hand, there was DC/OS, a stable platform built around proven technologies like Mesos and Marathon, backed by a well-funded startup called Mesosphere.

DC/OS sported a friendly user interface, a stable feature set, and a commercially supported enterprise version. On the other hand, there was this plucky upstart called Kubernetes. The internet was abuzz because "it came from Google", but what we saw was an immature feature set, rapidly changing APIs, a steep learning curve, and so-so usability.

Needless to say, we chose DC/OS. Fast forward a couple of years, Mesosphere rebranded itself as D2IQ, dropped all support for DC/OS, and turned into yet another Kubernetes vendor. With container orchestration being a central feature of the platform, we faced quite a challenge to migrate everything over to Kubernetes.

Luckily, we'd seen the writing on the wall for a couple of years by then, and had introduced sufficient internal abstraction layers to allow us to swap out DC/OS for Kubernetes without API impact for our tenants[2].

The moral of the story—*Plan for change because it's the only thing you're ever certain of.*

1 https://klarrio.com/project-talking-traffic/
2 Read our post "Migrating from DC/OS to Kubernetes: A Deep Dive into The Challenges and Opportunities"

YOU CANNOT RETROFIT SECURITY.

In the connected world we all live in today, security is paramount. But legacy systems originally designed without security in mind are very hard to retrofit. Why is that? Well, Michael Nygaard (the man who popularized the concept of Architecture Decision Records[3]) defines architecturally significant decisions as:

“those decisions that influence the structure, non-functional characteristics, dependencies, interfaces, and construction techniques of a software system.”

From this definition, we can see why the introduction of security in a software system has repercussions across the entire architectural board. For instance, you may want to alter the structure of your software system to better separate responsibilities and ensure you follow the Principle of Least Privilege.

There’s a whole new layer of non-functional characteristics to take into account. What’s more, existing non-functionals like latency and throughput may be impacted when TLS handshakes increase connection setup time and encryption strains processors more than before.

You may also be forced to switch out dependencies if the ones you have right now don’t measure up to your overall security standards. The interfaces of your software project, both external-facing and internal inter-component interfaces, will change due to the introduction of authentication, authorization, and access control.

And the way you construct software overall is impacted. Vulnerability mitigation, supply chain security, and other concerns may trigger a full-scale redesign of your software build and delivery pipeline.

In other words: bolting “security” on top of a complex brownfield software project is akin to smearing lipstick on a pig. It may look marginally prettier, but it’s still the same old pig underneath.

If you want to do security right, you either have to take it into account from the get-go, or you’re looking at a long-term, carefully planned, renovation project.

“Bolting “security” on top of a complex brownfield software project is akin to smearing lipstick on a pig. It may look marginally prettier, but it’s still the same old pig underneath.”

3 https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions

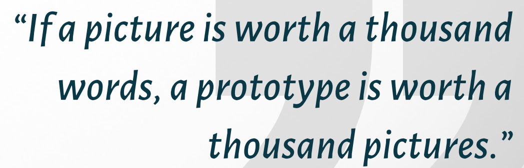
ELEGANCE DOES NOT WIN YOU PRIZES. VALUE DOES.

Klarrio builds software not for the sake of doing so, or because it's an art form. We build software because it delivers value to a business. Hence, our goal is to maximize said value regardless of what a specific project entails.

An architect's job is to do so over the mid-to-long term, by keeping the balance between quick wins and technical elegance. This ensures that technical debt doesn't build up over time, and the foundations of the system remain sufficiently modular to sustain constant evolution.

Architects and developers alike need to understand that "perfect" is often the enemy of "good". They must also keep each other honest. If an architect's design makes the implementation ten times more complex, developers must speak up and push back. Likewise, if architects see that developers are overcomplicating the implementation of a component, they need to step in and put development back on track.

Sometimes, you just have to deliver value. You have to sit down, stop talking about what you intend to do, and get shit done. It's as simple as that.



"If a picture is worth a thousand words, a prototype is worth a thousand pictures."

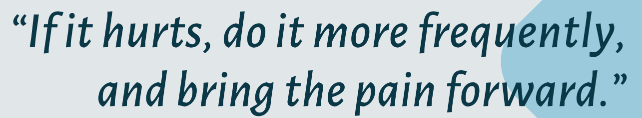
PROTOTYPES ARE A CONVERSATION PIECE.

There is this famous quote, apocryphally attributed to Henry Ford: *"If I would have asked people what they wanted, they would have said faster horses."*

Regardless of the provenance of the quote, it does bring home an important point. Requirements gathering is a tedious business. People are often very bad at articulating what they need and confuse what they truly require for what they think they want.

We have found that people are much more articulate if you put them in front of a prototype they can poke at and play around with. Our most valuable requirements often come from people pointing at a prototype and saying, "I want more of this, less of that, and this should work like so-and-so."

If a picture is worth a thousand words, a prototype is worth a thousand pictures.



— JEZ HUMBLE,
CONTINUOUS DELIVERY: RELIABLE SOFTWARE RELEASES THROUGH BUILD, TEST, AND DEPLOYMENT AUTOMATION

TO MOVE FAST, YOU NEED CONFIDENCE.

We have already established that large-scale software projects are developed in an ever-shifting context, with the forces of technical and business objectives in a constant push-and-pull. To remain relevant in such an environment, it’s imperative for iterations to be short, feature development to move fast, and new releases to be deployed frequently.

But to move fast you need confidence. Confidence in the quality of the code produced by the engineering teams. Confidence that newly introduced or reworked features don’t introduce regressions or cause downtime. Confidence that the system is, and remains, secure.

The only way to make this work at scale is to automate every step of the release process, and to fully automate testing and quality assurance. If you want to release often, it’s no longer feasible to have an engineer prepare a release by hand. It’s also no longer possible to execute hundreds of manual tests on a QA cluster to verify that everything is still working. We have seen legacy systems where the QA process literally takes months to complete. This precludes doing more than a single release per year.

The idea of frequent releases is embodied by the CI/CD movement. Adopting CI/CD often gets conflated with adopting a CI tool (Github Actions, Azure DevOps, CircleCI…), but CI/CD is more than a tool; it’s a mindset. For continuous integration to work, you need automated testing. And to get good automated test coverage, testability has to be a first-class citizen within the architecture. It’s not enough to build a system first and figure out testing later. Systems should be designed from the ground up to be verifiable through automation.

Test-driven development (TDD) addresses part of this need by encouraging developers to write tests before code. But TDD often focuses too narrowly on unit testing. True architectural testability extends much further—to system, integration, and end-to-end testing. Large distributed systems are notoriously hard to retrofit for better testing because of backward-compatibility constraints and the cost of broad architectural changes. To avoid this trap, architects must plan for testability early.

Once systems are architected for testability, automated testing becomes easier and more reliable. Better testing enables faster, safer releases, which in turn frees capacity to improve both product and testing infrastructure. This creates a *flywheel effect*: automation drives confidence, confidence enables velocity, and velocity reinforces continuous improvement.

CLOUD-NATIVE ISN'T JUST A BUZZWORD. IT'S A PHILOSOPHY OF DESIGN.

Architects often talk about making software cloud-native, or adopting cloud-native practices. Today, nearly everything is labeled "cloud-native", which makes it easy to dismiss the term as just another buzzword.

There is more to it than that. On the one hand, the cloud has created new opportunities like pay-per-use pricing and elasticity. On the other, it magnified the fact that hardware and networks are fallible and will fail in production, an inconvenient truth that was often swept under the rug in traditional data center architectures.

Together, such problems and opportunities pushed designers toward new concepts like designing for automation, containerization, stateless services, API-first design, horizontal scaling and resilience through multi-instance deployments. These are all just good engineering principles that are applicable beyond the cloud.

Take containerization, for example. Containers have become the ubiquitous method for packaging, storing, and deploying applications. Containerization solves the "it works on my machine" problem by packaging an application with all of its dependencies, creating a consistent and portable environment that runs the same way on any machine. It's a useful problem to solve, no matter where the application is running.

Adopting cloud-native principles generally leads to a cleaner architecture with its focus on communicating through well-defined APIs and resilience. These are principles that not only make it easy to deploy the application in the cloud, but also make it easier to test, scale, and adapt.

DATA HAS GRAVITY.

The true treasure chest for most companies isn't their arsenal of software systems, but rather the data these systems collect and process. As data volumes grow, the data itself starts exerting a peculiar kind of gravitational pull.

On the one hand, most storage-solution vendors, both raw byte storage and database solutions, have a knack for making it easy to load data into the system but slow or costly to move it back out wholesale. Once you've chosen a solution and stored a non-trivial amount of data in it, you're effectively locked in.

Your only option is likely to pour more data into the system, and a critical mass of data tends to pull in even more data as if it were an informational black hole.

On the other hand, storage is relatively more ubiquitous and cheaper than bandwidth. To process data in a way that is fast and cost-effective, compute gets moved as close to the data as possible. Data's gravitational pull brings compute into close orbit.

So, it pays to think about data strategy in advance, and the use of open formats and open source technologies with broad ecosystems helps keep that gravitational pull in check.

Users of a software system mostly care about what that system can do. As a designer, however, it's almost as important to be upfront about what a system can't do.

The idea of information hiding, or encapsulation, is well-known and widely applied in object-oriented programming. Classes provide a public interface that specifies how one can interact with instances from that particular class, while at the same time hiding all of the implementation details.

This gives great freedom to maintainers. As long as the public interface remains intact, the internals (data structures, algorithms…) of the class may vary from version to version. Without this kind of shielding, users of the class would start to rely on implementation details and undocumented behaviors. This leads to a calcification of the code base, where any change, even a bug fix, will break some user's code.

We have found that this same principle should be applied at all levels of granularity in system design. Components of a complex system should have well-defined APIs that are maintained in a backward-compatible fashion, and they should take pains to seal off the internal implementation and data stores from the rest of the system.

In addition, the system itself should do the same. It should have a well-documented public API (and perhaps UI), and explicit documentation about its intended uses and behaviors. Architects should enforce these boundaries with the system's users, preferably by designing the system in a way that ensures the internals are properly shielded.

The more complex a system becomes, however, the more these internal details will leak out in some way or form. External users and automations written against a system may implicitly rely on certain timing aspects, or the order in which certain things happen in the system, even if they aren't part of the public API.

To safeguard the possibility for evolution and the ability to move fast, architects should stand firm and educate users on the system's supported behaviors, thus creating the breathing room needed to allow for evolution.

03

PRINCIPLES.

We have codified the above insights into a set of principles. These principles are the north star for Klarrio's architecture and development work, keeping us from getting mired in the morass of overengineering, or mauled by any monsters lurking in the unknown.

PROTOTYPE EARLY. PROTOTYPE OFTEN.

Modern software systems are inherently complex. What appears on the surface often hides layers of intricacy and *unknown unknowns*. When faced with such complexity, the natural inclination of organizations is to create more process—longer design documents, additional review meetings, extended discussions.

In our experience, this does little to lower uncertainty, while at the same time drastically reducing agility. There is an alternative: experimentation in the form of *prototypes*.

A prototype starts with a hypothesis—an idea, a potential solution, or a new technology that might fit the problem at hand. Rather than debating every possible advantage and risk in meetings, teams can short-circuit the discussion by simply testing the idea. Build a quick prototype. See what happens.

"Modern software systems are inherently complex. What appears on the surface often hides layers of intricacy and unknown unknowns."

PROTOTYPING AS A PRACTICE.

Prototyping isn't a one-off activity. It's a habit, a muscle that strengthens with use. The more often you build prototypes, the faster and cheaper the process becomes, and the more valuable it is as a decision-making tool. Mature engineering organizations prototype constantly, not because they are indecisive, but because they recognize that learning fast is cheaper than planning long.

It's important to distinguish a prototype from a minimum viable product (MVP). The two are often confused, but they serve very different purposes.

- A *prototype* is built to answer a specific question. It exists to reduce uncertainty.
- An *MVP* is a first usable version of a product intended for end users.

An MVP might take months to deliver; a prototype might take a day, a week, or a sprint. The difference in scope and purpose is critical: Prototypes are about *learning*, not *launching*.

"Good architecture is never about predicting the future perfectly. It's about planning for change and learning fast enough to adapt when the future arrives."

THE LASTING VALUE OF A PROTOTYPE

Although prototypes are meant to be discarded, they remain valuable long after the experiment ends. If the prototype succeeds, it provides a head start on implementation: You've already explored the tooling, validated the approach, and uncovered practical constraints. Much of the exploratory work like understanding APIs, libraries, or infrastructure behavior is reusable knowledge.

If the prototype fails, it's still a successful step in learning. Knowing that a technology does not fit your needs is invaluable. It saves time, money, and frustration later. Failed prototypes also deepen understanding of the problem space. By manifesting the unknown unknowns early, they inform better architectural choices and future experiments.

PROTOTYPE TO ACCELERATE ARCHITECTURE

Prototyping aligns closely with a broader principle we also believe in—iterate quickly and iterate often. A well-designed prototype replaces lengthy architectural debates by providing concrete evidence, or it can serve as the input for those discussions. There are no rigid rules. Sometimes you prototype to explore, sometimes to validate, sometimes to communicate an idea. What matters is that you stay agile in the architecture.

Good architecture is never about predicting the future perfectly. It's about planning for change and learning fast enough to adapt when the future arrives.

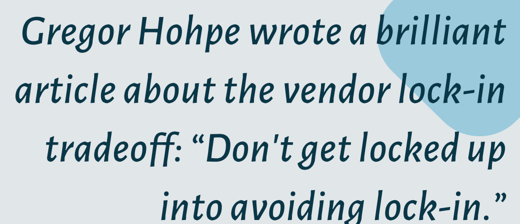
CONTROL THE COST OF LEAVING.

One of our core tenets is that you should control your own destiny[1]. From an architecture standpoint, this boils down to ensuring you don't get locked into specific vendors or solutions.

The relationship with your vendors should be healthy and mutually beneficial. It shouldn't resemble a hostage situation. To keep your vendors honest, it's important to have a way of walking out (and to be constantly aware of the cost of walking out, so you can keep it under control.)

As with everything in life, there is a balance to be struck. Jealously maintaining total independence of vendors is a costly endeavor. And going all-in on a vendor's offerings may be tempting because it's cheaper and faster in the short term. But when all is said and done, the cost of leaving may become unacceptably high.

The sweet spot in this trade-off is different for each and every one of our customers. As a rule, though, we prefer open source solutions (even when hosted as-a-service by a vendor) and open data formats over their proprietary cousins.



Gregor Hohpe wrote a brilliant article about the vendor lock-in tradeoff: "Don't get locked up into avoiding lock-in."

1 **Read Klarrio's Manifesto :** https://klarrio.com/manifesto

CONTROL THE COST OF STAYING.

Even if you have a healthy relationship with your vendors (and you're paying a fair price for the services they offer), you still need to think about the cost—especially in cloud contexts, where services scale elastically and self-service capabilities empower teams to rapidly spin up new infrastructure and services. Someone needs to keep an eye on the treasure chest.

Just like you need performance and error monitoring to ensure your services remain fully functional, cost monitoring is a must if you want to make certain your services remain economically viable.

On multi-tenant platforms, cost monitoring segues into the related-but-separate subject of billing. In our view the only true incentive for company divisions to control the costs they incur on a shared platform is to present them with an itemized bill.

Werner Vogels, CTO of Amazon, has summarized his experiences and insights on cost monitoring and cost-aware architecture in "The Frugal Architect[1]".

1 The Frugal Architect: https://thefrugalarchitect.com/

DREAM LONG-TERM.
PLAN SHORT-TERM.

For every software system, there should be a long-term vision that encompasses the breadth of functionality and use cases this system should offer. Given our assertion that change is the only constant, we expect this vision to be a living document, and we also expect it to be different in five years' time. So, it simply doesn't make sense to make a five-year plan to build the original vision because it will be obsolete by the time it is built.

As with many things in life, the 80-20 rule applies here. You can realize 80% of the value of the system with 20% of the features. That's why we advocate for an iterative development style where short-term goals are set in agreement with product management, and value is delivered in production as soon as possible. This approach may have been sold to you in the past as "Agile", "Scrum" or "SAFE", but those are just ways to formalize common sense.

Our approach to architecture supports this way of working. We maintain an overall system architecture definition that aligns with the goals in the (current version of) the product vision. Changes here are slow and deliberate; however, at the lower-level software architecture layer, things may change at a much faster clip.

ARCHITECTS ARE ENABLERS, NOT BOTTLENECKS.

Moving fast in the development of a complex software system requires a deep level of trust in the actual developers. If architects are involved in every single design decision, they become gatekeepers and bottlenecks that inevitably hinder the system's pace of evolution.

Instead, we feel that architects should position themselves as enablers of the development teams. Their primary role is not to tell developers what to build or how to build it, but to create a set of boundary conditions and empower developers to make their own decisions within the given boundaries.

Of course these boundaries start with a thorough understanding of business goals and an overall system architecture. In addition, architects should prepare clear sets of guard rails and requirements, both functional and non-functional. Ideally, the latter are embodied in a set of fitness functions that allow developers to constantly judge whether their implementation is still in line with the requirements.

The further you go down the abstraction ladder, however, the more the balance of power should shift: Developers become the decision-makers, and architects have a supporting role, offering advice, expertise and broader context whenever decisions need to be made. The final decision lies with the developers. They have to live with the consequences of their decisions on a day-to-day basis, so it's only fair that they get the final say in those decisions.

For greater insight into modern software architectural processes, please read "Facilitating Software Architecture" by Andrew Harmel-Law.
We consider it a must for anyone truly interested in the topic.

BE BRIEF TO BE RELEVANT.

In our discussion of the ever-changing context of a long-running software development project, we still haven't discussed one important source of change—the people working on the project.

Over a multi-year span the architects and developers in the project will inevitably move on and be replaced. If you aren't careful, institutional knowledge about the goals and design of a system will only exist in the heads of one or two veteran developers who remain on a project for the long haul.

This is obviously not a healthy situation. The key to avoiding it is to document all design decisions, so that newcomers can read up on the history and design ethos of the project.

Extensive documentation comes, however, with its own set of costs and tradeoffs. For one, documentation tends to age much faster than code. After all, for code there is a set of tests that constantly keep it in shape, but there is no automated way to ensure the documentation is still relevant and up-to-date with the latest design.

As a rule, documentation tends to become impenetrable over time. If you're a new joiner on a years-old software project, you're typically presented with a wiki full of documentation, most of which is either out of date or describes experiments that never made it to production. What's more, searchability is low, and even if you do manage to find something through search, there's a 70% chance it's outdated.

Lastly, the dirty little secret of technically minded people is that they don't like to write. It costs time to properly structure your thoughts and present material in a logical order. So, if every design decision must be accompanied by a 20-page paper on the necessity, design decisions, and impact of a given decision, we're inadvertently encouraging developers and architects to "forget" documenting their steps.

In short: We advocate for a style of documentation that emphasizes brevity and structure over completeness. It's far better to have a limited set of up-to-date, searchable documents that capture the essence of past decisions than to have a sprawling set of outdated references expounding vague architectural principles and considered-but-abandoned design dead ends.

It's better for design decisions to be captured in the form of Architecture Decision Records, in the style described by Harmel-Law's Facilitating Software Architecture book. These ADRs must be numbered sequentially and have a descriptive title and a clear indication of status (proposed, active, superseded).

Position papers that advocate for a larger design change should be limited to five pages. If they're longer, there is a very good chance they won't be fully read or understood by the target audience.

SECURITY IS FOUNDATIONAL.

The EU CRA act requires software to be “secure by design”, but even without regulatory requirements, software security is just good business sense. For Klarrio, this is so fundamental that we have written an entire whitepaper[1] on the subject.

No matter how fast we want to move things into production, and how much we believe MVPs bring value, there is one big non-negotiable—nothing goes into production without proper security measures in place. Every feature we implement is subject to a systematic threat-modeling[2] exercise.

Threat modeling only takes you so far, however. Security is a very wide-ranging subject that spans the gamut from the software supply chain (asset acquisition, build and deploy) over hardened configurations and penetration testing, all the way to good old-fashioned design (zero-trust, identity management, access control…).

To make a very critical story short: Every design decision, large or small, must take security aspects into account.

1 Read Klarrio’s Security paper: https://klarrio.com/manifesto/#Klarrio-secure-by-design

2 https://owasp.org/www-community/Threat_Modeling

04

CONCLUSION

Complex software systems are never a one-and-done story. They require constant maintenance and evolution to remain secure and relevant to the organization's business goals.

The architecture process plays a pivotal role in enabling this evolution. Done wrong, it becomes the spanner in the works. Done right, it becomes a key enabler and helps everyone involved in the project to move fast and build things.

ANNEX

05

ACRONYMS

- **ADR** — Architecture Decision Record
- **AI** — Artificial Intelligence
- **API** — Application Programming Interface
- **CI/CD** — Continuous Integration/Continuous Delivery
- **CRA** — Cyber Resilience Act
- **GDPR** — General Data Protection Regulation
- **MVP** — Minimum Viable Product
- **NIS2** — Network and Information Systems Directive 2
- **QA** — Quality Assurance
- **SAFe** — Scaled Agile Framework
- **TDD** — Test-Driven Development
- **TLS** — Transport Layer Security

HYPERLINKS

- Streaming data platform for smart traffic in the Netherlands: https://klarrio.com/project-talking-traffic/
- Migrating from DC/OS to Kubernetes: A Deep Dive into The Challenges and Opportunities: https://klarrio.medium.com/migrating-from-dc-os-to-kubernetes-a-deep-dive-into-the-challenges-and-opportunities-c6349e538c1a
- Architecture Decision Records: https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- Klarrio Manifesto: https://klarrio.com/manifesto
- Threat-modeling: https://owasp.org/www-community/Threat_Modeling
- Gregor Hohpe wrote a brilliant article about the vendor lock-in tradeoff: Don't get locked up into avoiding lock-in. https://www.linkedin.com/in/ghohpe/
- Jez Humble: Continuous Delivery: https://www.goodreads.com/work/quotes/13558958-continuous-delivery
- Werner Vogels, CTO of Amazon, has summarized his experiences and insights on cost monitoring and cost-aware architecture in The Frugal Architect: https://thefrugalarchitect.com/
- For greater insight into modern software architectural processes, please read: Facilitating Software Architecture by Andrew Harmel-Law: https://facilitatingsoftwarearchitecture.com/

ABOUT THE AUTHORS

Dominique Chanet - Director of Architecture at Klarrio

Dominique Chanet leads a team dedicated to delivering the best possible technical solutions for our clients at Klarrio. Prior to that he was a Software Architect in the CTO Office of Technicolor Connected Home (now Vantiva), and member of the Technical Steering Committee of the AllSeen Alliance, which stewarded the open-source AllJoyn IoT project. Dominique holds a PhD in Computer Science from Ghent University, Belgium.

Lasse Jacobs - Software Architect at Klarrio

Lasse Jacobs works on cloud-native and on-premises compute and data platforms. He holds a Master's degree in Computer Science.
Previously, he was a software engineer at Oqton, a Ghent-based startup, where he worked on a large fleet of microservices in a multi-cloud Kubernetes environment.

Method of contact: info@klarrio.com

This document is edited by Klarrio.

Due to the rapid development of related technologies in the streaming industry, this document is only for reference and cannot be used as a basis for investment research or decision-making.
All statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied. We may supplement, correct and revise relevant information without notice, but does not guarantee immediate release of the revised version. All statements, information, and recommendations in this document do not assume any responsibility for any direct or indirect investment profit and loss.

This document is an intellectual property of Klarrio. No part of this document may be reproduced or transmitted in any form or by any means without prior written consent. If any content of this report is released by any other party in the form of reference, Klarrio should be attributed as the source. Any citation, deletion and modification shall not violate the original meaning of this report.

For any questions or suggestions, please contact: info@klarrio.com

CONTACT US

- BELGIUM
- NETHERLANDS
- GERMANY
- SPAIN
- UNITED STATES

info@klarrio.com

www.klarrio.com

Klarrio specializes in large-scale and real-time data processing implementations. We offer deep expertise in cloud-native and hybrid solutions. But more importantly, we have the experience you need to understand where you've been, where you are today, and how best to achieve your goals going forward.

We're much more than a software solutions provider. Klarrio offers proactive and sustainable open source innovations with no vendor lock-in, all while working as a trusted partner who ensures you always remain in complete control of your own data.

Control your destiny. Contact us today.

Copyright © 2025 Klarrio™ BV- All Rights Reserved.

GENERAL DISCLAIMER
The information in this document may contain predictive statement, including but not limited to, statements regarding data security, future financial results, operating results, and new technologies. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purposes only, and constitutes neither an offer nor a commitment. Klarrio may change the information at any time without notice, and is not responsible for any liabilities arising from your use of any of the information provided herein.

Klarrio
STREAMING AHEAD