Domain Decomposition at Klaviyo: Divide to Conquer

Author: Fernando Salazar

Claps: 193

Date: Jun 20

I often share this quote from C. A. R. Hoare with my teammates:

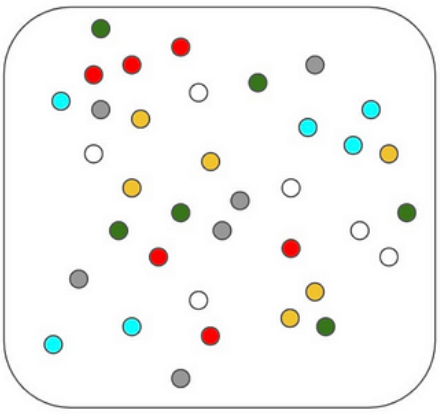
> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

Klaviyo being a fast-growing and ambitious tech company, it should come as no surprise that some of the code we have created is more on the complicated than the simple side. This is not just an issue of aesthetics. Complex code leads to *reduced velocity of improvement*, as it becomes increasingly difficult to make additions or changes that do not have unexpected side effects. It also creates *increased operational overhead*, as the different areas of the system cannot be debugged or tested in isolation.

Our answer to this is *domain decomposition*, an architecture process to transform our overall codebase from one of myriad, untracked dependencies into one composed of well-bounded domains that intercommunicate through clear interface contracts. In this post I want to share with you the process we used to get started on this journey, our results and learnings from the first iteration, and some going-forward thoughts. Letâ€™s begin â€¦

Theory

This picture shows our problem:

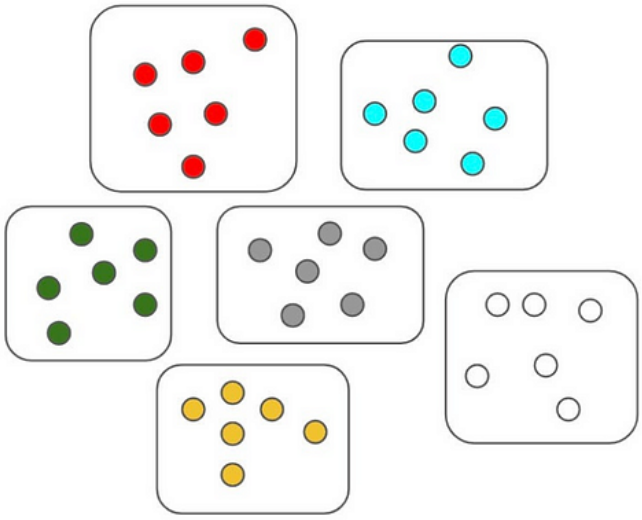


The colored dots are what we call *capabilities*, discrete technical outputs or steps. Example capabilities include:

- Render a personalized email payload.
- Render a personalized payload for SMS or Push.

- Transmit an SMS message.
- Schedule the sending of campaign messages using Push channel.

In the above picture, notice how capabilities of different colors are distributed across the entire codebase. This portrays how in many cases similar capabilities are implemented in places very â€œdistantâ€ from one another. Itâ€™s hard to see what depends on what, or where to take action to add a new capability. Our goal is to have a structure that looks like so:



Here similar or related capabilities are organized into bounded domains. This would allow these areas to better evolve or improve on their own, without affecting other areas. The net of our process:

1. Identify all the *capabilities* offered by our system.
2. Determine which capabilities should be grouped with others into *domains*.
3. Finalize those boundaries in *RFC documents* that provide the essentials of the interfaces and show how the most important use cases are supported in the transformed design.

Practice

Compared to say Amazon, Klaviyoâ€™s engineering organization is still small. Even so, the potential scope of the total analysis and refactoring work for us was considerable. The subject codebase is approximately one million files across >100 repositories, while the team working on that is over 250 engineers. Trying to propose an improved structure for all that, at one time, especially without having proven the process, would be a risky undertaking.

We therefore decided our first pass at this would be confined to a single *pillar*, a Klaviyo organizational unit similar to a division in other companies. The thinking was that by working on domain decomposition for one pillar, vs. everything, we would have a large enough project to give meaningful results and learnings, but not so big we would be affecting progress across the entire company. The subject pillar was *Channel Infrastructure* (where I work as a senior engineering manager), a team of about 60 engineers. Channel Infrastructure owns mostly backend code around the different Klaviyo means and methods of sending marketing and informational messages.

We enlisted a fairly large group in this effort, about 35 people overall, including both engineers and engineering managers. There was some concern this number would be too unwieldy to make progress. I had (and still have) a strong belief that we needed widespread investment in the

process and the result. I did not want a model where all the decisions were made by a very small group of select leaders. More about this later.

After we had our kickoff meeting that presented the theory and the plan, we hit our first hurdle: What is a â€œcapability,â€ really? Turns out that most of our team thought of these as *business outcomes*, for example *performing* an email campaign. This perspective is not surprising, since all of us had long viewed our systems from a customer-facing, application perspective. What we needed was to identify the specific *actions* that lead to those aggregate results. Hereâ€™s the definition of capability we arrived at:

- A unit of function offered by a domain.
- A discrete output, processing step or state change.
- Is not just an optimization of another capability.
- Is not just a transport or relay.
- Is often something replaceable by a new implementation.
- Is not a business outcome, but a technical result.

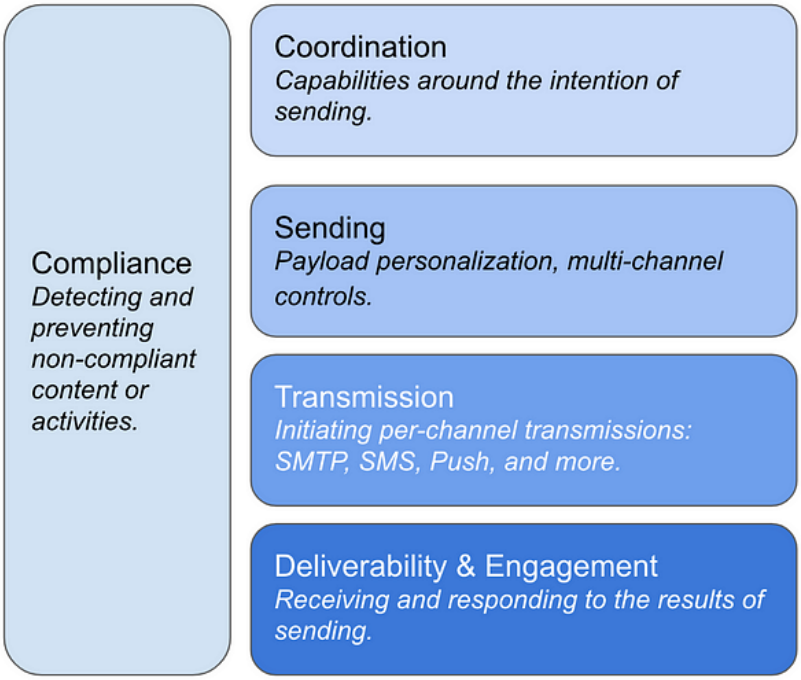
So to refer to our previous examples, rendering a message payload *is* a capability, because it is a discrete output. However enforcing regulatory compliance for sent messages *is not* a capability; rather that is a business result requiring the combination of multiple specific capabilities, like check â€˜spaminessâ€™ of a message and flag an account for disablement.

Final point here, capabilities are fundamentally *outward facing* â€” they are services offered by a domain. Of course internal to any domain there are many components and processing steps; these *are not* capabilities. It is important to *hide* all that from other domains, the focus needs to be on how the coarse-grained parts of the system interoperate.

We identified >80 capabilities. Iâ€™m sure there are actually a few more, but we thought it better to get onto the larger part of the project, namely: organizing the capabilities into domains.

Making it work

As I noted earlier, there were more than 35 team members involved in this process. Thatâ€™s a lot of people, much more than could productively collaborate in meetings. Our solution to this was to form *working groups*, sub-teams that could practically address parts of the larger problem. The overall responsibility of our pillar in Klaviyo is the sending of marketing and informational messages using a multitude of channels. We were fortunate in that essentially everyone in our project was familiar with the conceptual layering of that problem space, as shown here:

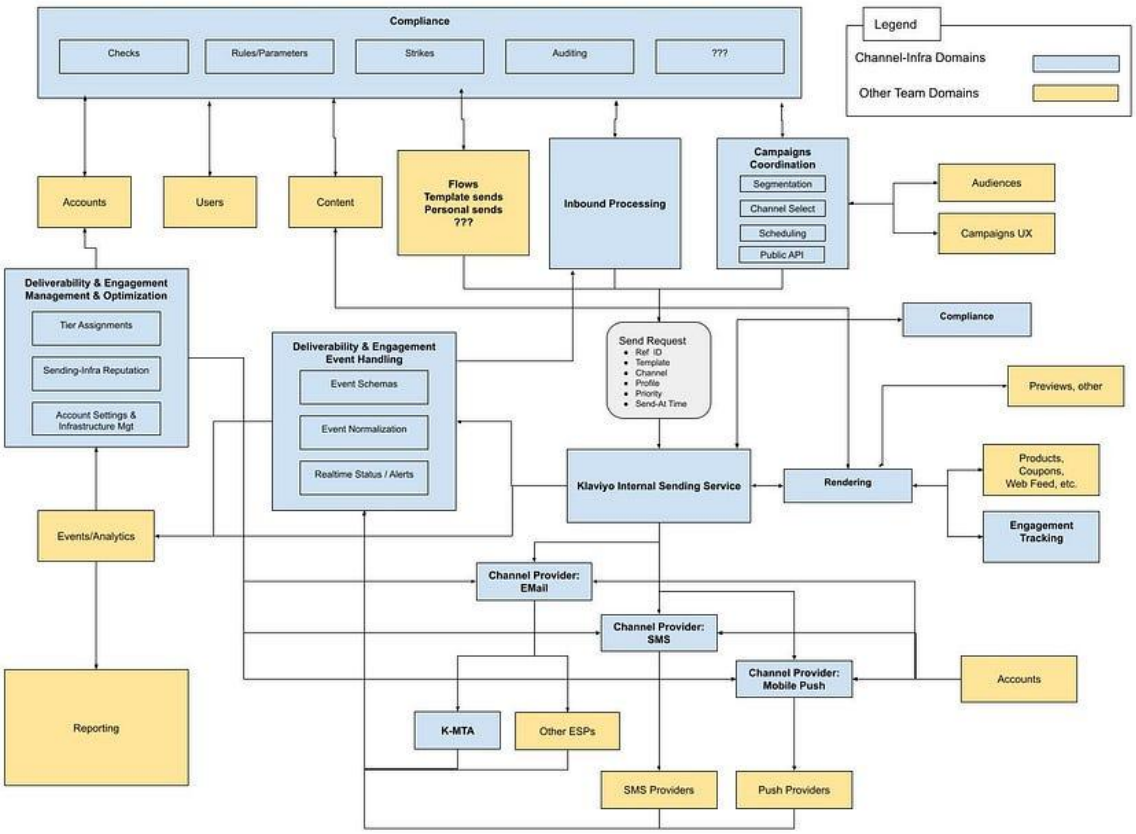


A marketing campaign is a *Coordination* function, in that what is important there is to know, for example, for which specific recipients I wish to send specific content at certain times. *Sending* comprises things that make those intentions happen, as well as some cross-channel or cross-coordination functions like overall message throttling. *Transmission* is about performing the actual protocol sends, while *Deliverability & Engagement* is about the results of those sends â€” how many emails or texts were clicked, how many bounced, and so on. And finally *Compliance* is orthogonal to all those, in that we may need compliance tests at any of these layers.

As I said, everyone on the project was familiar with this. The new thing was, *all of our teams were used to working on all layers*, based on their channel affinity. Breaking those channel â€œsilosâ€ enabled people to see how by making these strong separations we could get economies of scale. When you think about it, why should a marketing campaign â€œcareâ€ about the internal details of how a payload is rendered, or how SMTP is used, or even what the transmission channel is? No reason â€” in fact what we want is *decoupling* of those concerns so, for example, we could totally replace the SMTP transmission function without worrying whether that would affect campaigns.

Our results and the futureâ€¦

Our goals for this project were: *identify capabilities*, *organize them into domains*, and *write RFCs that document the interfaces for each domain*. We achieved that in three months. The picture below shows the domains:



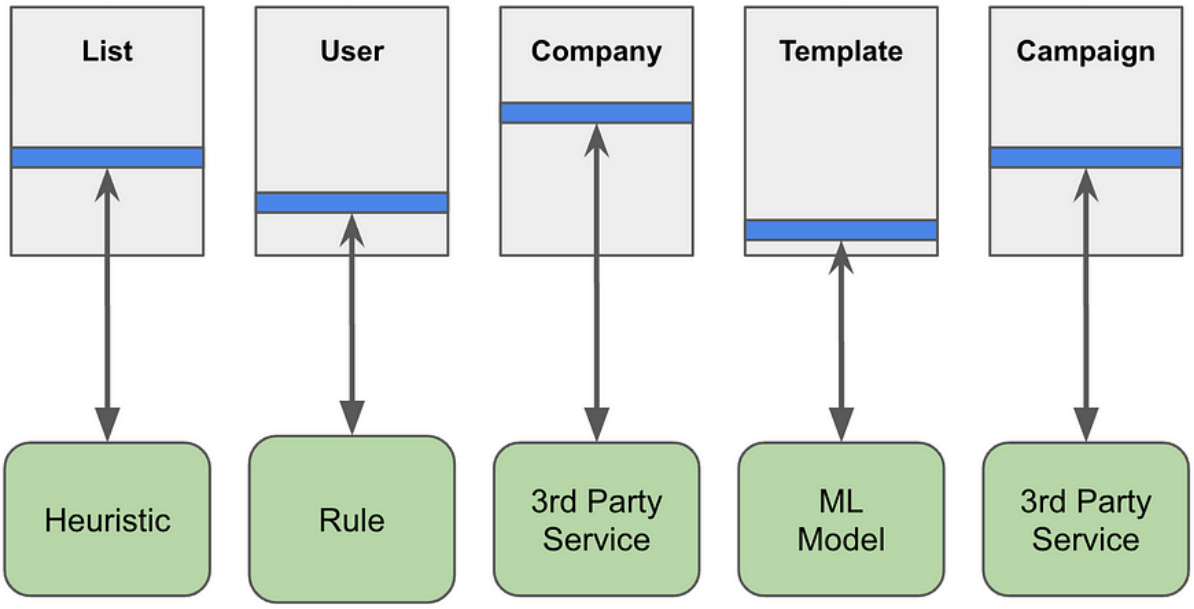
The blue boxes are the domains we identified in our pillar, the yellow boxes are dependencies external to our pillar. Readers are of course invited to look at the names of things (sorry for the tiny type) and form your own ideas on the approach. But Iâ€™d like to offer some of the tests we had to pass as evidence of the soundness of this design:

- *Are the interfaces for every domain understood?* Itâ€™s all very well to propose a box named â€œCompliance,â€ but until that box has an interface, thereâ€™s little we can determine about the usefulness of that box. We focused on the essential semantics of the interfaces, not detailed Python or REST APIs. Some examples:

Operation	Description	Input	Output
`create`	Initiate a Compliance Check (sync/async) or Track Event	`CheckType`: (e.g. `SendEmailCheck`, `TemplateCheck`, `TrackContentCreation`, `TrackProfileCreation`) Options: (e.g. `sync`/`async`, `company_id`, `group_id`, etc items containing to specific `CheckType`)	**CheckResult** `ULID` `CheckStatus` (e.g. `FAILED`, `IN_PROGRESS`, `SUCCESS`) `CheckType` (e.g. `SendEmailCheck`) Timestamp Request Metadata
`get`	Fetch `CheckResult` (sync). Used as one-off retrievals of a given Compliance check.	Check `ULID`	**CheckResult** `ULID` `CheckStatus` (e.g. FAILED) `CheckType` (e.g. `ListOverlapCheck`) Timestamp Request Metadata

- *Can the proposed interfaces support every business function where our code plays a part?* Once we had the first strawman version of this, we started the thought experiment work to answer questions like: How will an SMS campaign actually send? How will campaign options like Smart-Sending or Recipients at Send-Time be implemented? This was the biggest part of the final work and involved some â€œrebalancingâ€ of capabilities between domains.
- *Are the interfaces as minimal as possible?* A practice in our current system is that certain core objects get used at all layers of the application. One of these is a *Message* object that includes information about templates, subject lines, A/B tests and more. But the function to render a payload really only needs the Template â€” why should we send it the full Message? It was our intent to as much as possible follow the Law of Demeter which holds (among other things) that information passed between components should only be the minimum necessary to perform the desired function.

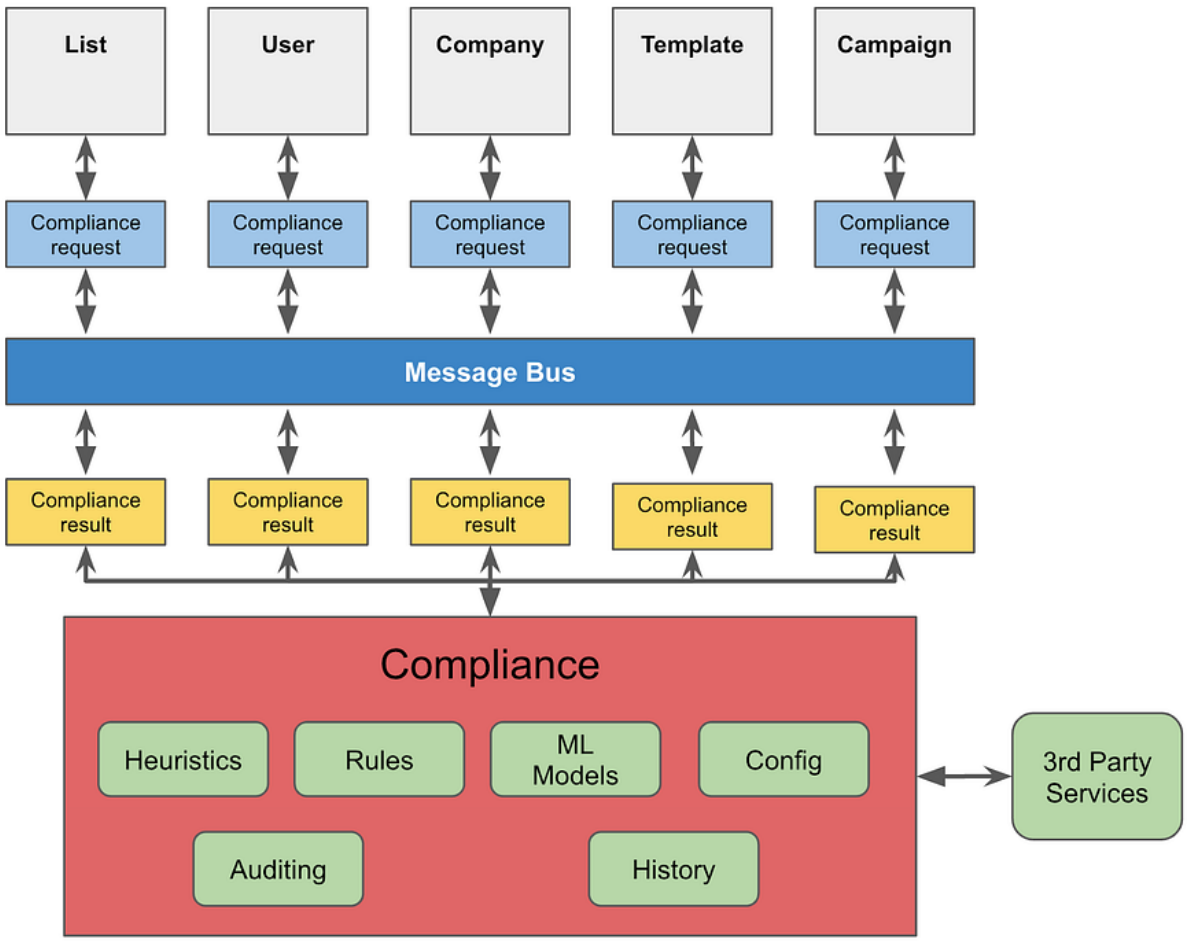
Another positive bit of evidence is in how teams are now embracing the results of this effort and working to implement these approaches in the context of new projects. A major example is in the compliance area, where Email and SMS Teams are collaborating on a major update to compliance for SMS messages. That work is beginning from the standpoint of a well-bounded Compliance domain, with dedicated operational logging and administration of rules and settings, all in service of making compliance determinations in a way that is independent from the implied enforcement actions. I think this is a cool improvement, so let me show a little more detail on what that looks like. Hereâ€™s a representation of of how compliance in our system works today:



The gray boxes, List, User and so on, are entities that have to be Compliant. The current implementation is that an engineer specializing in compliance inserted code â€” the blue stripes â€” to make a compliance check. That code typically invokes some service, like an ML model or a partner API, to perform that needed test. Then, based on the results of the test, the inserted code takes some action.

This is a fragile approach. All these gray-box entities are constantly changing as we implement new features in these areas. All it takes is a change in internal order of operations to move the compliance code to the wrong place in the object lifecycle.

Hereâ€™s the picture of the new approach:



The main points in the new design are:

- The entities that are subject to compliance must emit *compliance requests* at appropriate points in their lifecycle.
- These requests are handled by a *single, unified engine*, that can easily share code between request handlers, and can update handlers without any subject entity having to participate.
- The majority of these interactions are asynchronous, using a *message bus*. Synchronous handling of requests is supported when use cases require.
- It is the responsibility of the initiating entity to act on the *compliance result*, for example disabling the template, etc. that failed its check.

Compliance is not the only place where implementation progress is happening. Other teams are working on RFCs, prototypes, and limited-availability implementations of the designs shown in the diagram. The full scope of the work we defined will stretch to the end of 2024, but the momentum to make that happen is there and growing.

Lastly, it is both satisfying and exciting to see how the results from our Channel Infrastructure effort is leading to plans and projects across the rest of Klaviyo engineering. Other pillars are adopting the capabilities/domains model and applying it to their areas. And our Architecture Review Board (ARB) is now driving multiple projects around standards and tooling that will make the realization of domains and interfaces easier and consistent across all our teams.

Thatâ€™s the opening chapter of the domain decomposition story at Klaviyo, the first of several more to come. But already we are seeing that ***with software, the way to conquer is to divide things up in the right way***.