Courier

The Product Manager's Guide to Building Product Notification Systems

www.courier.com

Contents

Introduction

So your CTO has just handed you a project to revamp or build your product's notification system. It seemed like a simple and straightforward project, but you started doing research and realized that not only is the process pretty complicated, there's not a lot of information online on how to do it. After all, companies like LinkedIn, Uber, and Slack have large teams of over 25 employees working just on notifications. But smaller companies don't have that luxury - so how can you meet the same level of quality with a team of one? This can certainly be overwhelming, which is why we've created this ebook to guide you through building the best notification system for your company.

User Requirements

It's crucial that before building a notification system, you should know the user requirements for your fellow teammates who will be creating the notifications for your end users. Understanding these teammates' personas will help you to build a more effective product with a better user experience.

What Is a Notification System?

A notification system is a collection of services (templates, provider integrations, routing logic, preferences, logging, etc.) that make it possible to quickly and easily create clear and direct communication between an app and its users. This clarity of communication generally involves a myriad of channels, including email, SMS, push notifications, etc. that allow the app to reach each user with the best possible user experience.

A well-built notification system removes complexity from the process of creating each notification, which allows for a consistent experience across products and teams. This also provides a centralized hub for notifications across the organization, thus making monitoring and analytics more accessible.

Depending on your company's product, a notification system can work with different use cases. You can use a central notification system to alert your end-users about an incoming request, send messages about actions taken, inform end-users about product updates and upgrades or promotions, or even deploy account management notifications.

PM Requirements for a Notification System

A PM needs to understand the framework of the notification system so they can integrate it into other parts of the application or software. They are the ones who end up wiring up notifications for the myriad of applicable use cases, so it's important to build the system with them in mind.

Scalability and Reliability

Reliability makes it possible to avoid dropped messages. Even if many messages are coming in simultaneously or the system is at peak load, message delivery should be guaranteed. While there could be delays at peak load, you should be confident you aren't losing messages.

The system should also retry failed message deliveries by sending messages reliably over the network and try again if a message fails.

An organization will need to send varying volumes of notifications at various times, so the user using the system will not need to bother with auto-scaling the infrastructure. For example, an organization needs to send plenty of notifications when it has flash sales. At other times, during low-volume periods, it will need to send fewer notifications. The system should be able to scale up and down resource-efficiently as the volume of notifications changes and as an organization grows.

Abstracting Channels

In the absence of a central notification system, inconsistency among channels like SMS, email, or push notifications are likely to become an issue whether among fellow PMs, customer success, marketing, etc.

You can change the notification channel provider, whether AWS SES or Twilio for emails, in the notification service without changing the application code in any other products. Thus, the notification channels and providers will be abstracted and centralized instead of having the code sprinkled all over the application codebase. So if your company stops doing business with a particular provider, it can switch to a new provider in a few hours without impacting any other part of the notification service.

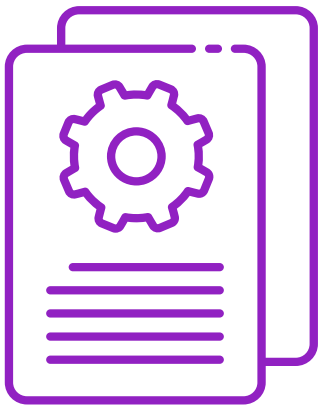
The uniform interface makes development simpler for each product and team by abstracting the different notification providers (email, push, SMS, etc.) so any employee can easily switch between different providers without rewriting the code.

Good Documentation

For other employees to use your notification system, you have to provide good documentation to understand how to use it. Internal documentation is an integral part of any system, as it educates and helps users to reference and know how to use your product. When building a platform for other teammates, you have to provide great documentation so that they have the tools to figure out any support-related issues.

Good documentation for a notification system should be an easy guide to help them get started. It should also provide a comprehensive reference for all the operations in the notification system. A developer integrating the notification system should not have to guess supported operations and parameters for the system.

The documentation should also be easily discoverable. Without knowing exactly where to go in the system, the developer should search for what they want and find it easily. Documentation should be accurate, consistent, available on-demand, and up to date. It should include examples, code samples, screenshots, and tutorials for more context on the system.



Intuitive APIs

Technical users need to send notifications programmatically. So, a notification system must have APIs to submit notifications for delivery. These APIs must be intuitive from various platforms so that the system does not constrain the implementation of other systems. Users should call the APIs from any programming language or platform, and the API documentation should also be available on demand.

Analytics

Notifications communicate with a target audience—so, system operators want to measure the performance of the notification system (and the impact of the notifications themselves) and collect information that can help the organization design its notifications better.

Integrations to Export Data to a Data Warehouse

The data the notification service collects is valuable but raw. So, analyzing the data further derives insights for your organization. Such analyses are often done in other systems where engagement events correlate with other data to better picture user behavior.

A notification system should support data exports in both human-readable and machine-readable forms. It can also integrate with data warehousing tools and export to them directly.

Interactions

Engagement with a notification is an essential metric for businesses, so the notification system should track such engagement. Tracking engagements are usually done by tracking link clicks and push notification opens.

For links, the notification system rewrites links in the notifications to go through itself. Each visit to the links logs an event in the system, then redirects to the original link. That enables the system to track clicks. The SDKs on the clients notice when a user opens a notification and record it for push notifications.

Operational Metrics

Information about the run-time behavior of the system is important for keeping the system running. Latency metrics and throughput metrics help to understand delays in the subsystems and the rate of notifications delivered. Queue length, together with service time and wait time (both latency metrics), can estimate delays and optimize the system further. These metrics help with capacity planning.

Support for Integrated Logging

When things go wrong, the system should let users gain insight into those issues. Technical logs provide such insight. For example, the logs show that although notifications were successfully submitted to the system, they were not delivered to the downstream provider. The users now know that it's not us; it's them.

Users should see detailed technical logs for errors that occur when a notification is not sent. Another example: the user should see that a message sent through SendGrid bailed because of an HTTP 401 error that says the API key is bad. Technical logs also show other vital details about the system's operations. The logs can help operators diagnose problems when they occur.

Support for Test Environments

A test environment allows the employee to simulate sending notifications safely. It is useful in continuous integration or staging environments where you need to run test code without sending notifications to actual customers or end-users.

Supporting a test environment enables rapid application development and also gives confidence to the team. The developers can write tests close enough to the notification system's workings rather than mocking out the system in their tests.

A test environment also allows the PMs to experiment and try out different parameters and operations to see their results without impacting customers. Without this, every interaction with the notification system is potentially dangerous, as it may send a notification to a customer. A system that does not support a test environment delays the development pace because PMs have to wait until production to try things out.

White Labeling

If your company has spanned multiple products or brands, the notification system should be able to deploy notifications to other brands' customers and change the branding and logos on the fly. White labeling makes changing over to new brands for sending notifications much easier. The newly acquired company can retain its brand image while switching to the existing company's notification system. For example, Twilio, Segment, and SendGrid (all owned by the same company) want to send notifications to all three software and change the branding and colors on the spot, depending on the product receiving the notification.

Non-Technical User Requirements for a Notification System

Non-technical users are the ones who only need a smooth user interface and user experience to use the notification system. Designers, content editors, customer success and support, and your marketing team do not interface with code, so you have to build to suit their needs as well. Let's look at some of the requirements for a non-technical user.

Usability

Usability is a non-negotiable requirement because your users need it to create and send notifications seamlessly. Ensure the interface is user-friendly so they can explore the system quickly. It should also not require a lot of onboarding and training to understand the system.

The users should be able to carry out their intended tasks efficiently (in the shortest sequence of steps possible). To achieve decent usability, choose task-based user interfaces over generic ones. Task-based interfaces are interfaces designed with a particular user action in mind. For example, logging: A customer support representative needs to find why a user is not receiving a notification and needs to be able to search for notifications to that user by email in the logs. The interface must:

- Clearly differentiate different kinds of logs and show the relevant information for each one through a specific identifier (in this case, email).
- Allow searching for logs involving the user with the specified email address within the time period that the user stopped getting notifications. This makes it easy to jump to the relevant log information while wading through a large amount of data.

Designing a Notification

Designing a notification is the most important capability for a nontechnical user who will not be handling any code. Creating content and efficiently designing its layout and branding plays a central role in the way a customer success manager, for example, might use your notification. The manager needs to be able to rebrand a new logo or update text within an email or SMS without engineering going through a sprint cycle.

In addition to creating a great UX for notification design, templates can help make it faster and simpler to design the notifications. These templates could provide a drag-and-drop editor to change content on the fly without redeploying code. A highly usable system should also provide ready-made templates and the ability for the user to to create new ones and customize them.

Historical Records

Non-technical users need to see some logs, although not as detailed as technical logs. Each log entry should contain information such as:

- the user that sent the notification
- the time
- the recipients
- the channels used
- the cost incurred if known
- the content of the notification

The notification system should also log notifications sent by other systems through API calls, not just those sent by human users. Besides logs that send notifications, the system should also log changes to access permissions. It is essential in notification systems that have role-based access control.

It is important to know when a new user has permission to access the system. The notification system should log the particular permissions granted, the user that granted the permissions, and the user that received the permissions. The system should also log an event when it revokes a user's access. Permissions granted to machine users, such as API keys and service accounts, fall under this category. These logs provide insight for non-technical users to understand their use of the system over time.

Role-Based Access Control (RBAC)

Role-based access control is a system that grants permissions based on roles defined in the system. It makes it easier to manage access across an organization and tailor such access to the roles of employees and departments.

For example, suppose we want to enforce the following rules:

- Only the marketing team can send notifications.
- Only the design team can change notification templates.
- Only heads of select departments can add new roles to the system.

With RBAC, you can create three roles: notification-sender, designer, and role-editor, respectively, for each rule. Users who take on these roles can perform them, and users who don't have them cannot perform them. Build the notification system in a way that is easy for small teams to use, and it should scale to larger teams and organizations, too, as they need it more.

One important part of designing an RBAC is the ability to compose larger roles from smaller roles. For example, it lets you create roles that delegate smaller functions to subordinates while granting more permissions to team leaders.

Analytics

Non-technical users need to see the collected data in an easily digestible way. The system should present data in a format that's easy to understand so even non-technical users can grasp key insights at a glance.

The system also needs to provide various views of the same information: aggregate statistics over different periods, various visualizations of the data, etc. They should answer the most common questions, such as what notification channel performs best for each type of message, by looking at a dashboard.

Moving Forward

Understanding the needs of the different personas that will be using your notification system is foundational while building a notification system to ensure that your hard work is meeting the needs it was commissioned for. But understanding the needs of not only your fellow team members from customer success, marketing, support, etc. will make your hard work more available and scalable.

Scalability and Reliability

There are many helpful tools for building a notification system, but it's no easy task, especially when reliability and scalability have to be taken into account. For a company to grow, it will eventually need to decide between the cost of building and maintaining its own system, or opting for the functionality and proven reliability of a third-party product. This is known as the classic build-vs-buy decision.

While the cost of purchasing a solution may be clear, the cost of building your own can be difficult to calculate. In this section, we cover building a scalable and reliable notification system in detail to give you an idea of the required effort.

The Keys to Success

Scalability and reliability are two distinct, yet interrelated aspects at the core of a good notification system. You achieve reliability when your customer receives all of your notifications without errors or duplicates. This means reaching your customer consistently and on time. Scalability is where your application can handle higher notification volumes as a result of your product's growth.

It costs time and money to improve a notification system's reliability. If you're still looking for product-market fit, it might not make financial sense to prioritize reliability when your resources might be better allocated elsewhere.

Once you find product-market fit and grow your user base, however, your notification volume will increase quickly. If you're growing fast, you might choose to invest more into fixing other critical parts of your SaaS application instead of improving your notification system's reliability. But you might jeopardize your product's growth if your customers don't receive your notifications due to errors, timeouts, or delays. If a problem-ridden notification system starts impacting the user experience, it's just as likely to impact your bottom line.

Scalability and reliability are both key considerations for any build-vs-buy decision. For example, when the feature management platform LaunchDarkly was making its own build-vs-buy decision, it had to consider its SLAs, SLOs, and SLIs as part of its investment in a notification system. It had recently closed its Series D funding, and substantial volume and load, compliance, reliability, and stability were all key factors in the decision-making process. LauchDarkly decided to go with Courier because the platform met LaunchDarkly's strict scalability and security requirements, provided necessary features, and fit seamlessly into LaunchDarkly's tech stack.

No Scalability or Reliability, No Growth

Scalability and reliability are two different aspects. But both become concerns if you want your company to keep up with a growing customer base. If you lack one or the other, you'll likely meet problems along the way.

If your notifications lack reliability, your brand's impact stands to lose. To a product like Slack, a delayed push notification does not have much utility. In Slack's case, timeliness is crucial to creating a real-time conversation between team members.

But even if you're not Slack, losing early users' trust in your notification system can slow growth. Your early adopters will be unlikely to recommend your product if they don't trust the way it works. Duplicate notifications represent another scenario that can be a turnoff for early users. Receiving duplicates of notifications frequently suggests to users that the product isn't stable enough to use, so early adopters might hesitate to share the product with their friends or colleagues.

So, what is the most common source of scalability and reliability issues in a notification system? Based on Courier's experience, it's the fact that notifications are rarely spread out evenly over time. The reality of unpredictable volume spikes requires an understanding of how to scale infrastructure to handle high volumes at a reasonable cost. If a system doesn't scale well to accommodate peaks, notifications will end up being processed and delivered beyond their relevance window. In the worst case scenario, an overwhelmed message queue can result in a system outage. In short, if you find your service growing and your notification system is not equipped to handle it, you are taking on considerable risk.

Moreover, a notification application needs to be kept available to its users while its code is replaced. If you designed your application without keeping that in mind, you face the possibility of extended downtime while you work on it. Downtime means your users won't receive notifications, and won't be engaged with your product. Ultimately, designing your notification system to reduce downtime saves you both time and money.

A system built without both scalability and reliability in its design patterns also risks frustrating and overworking your engineering team. Engineers on call risk getting burnt out if they have to constantly respond to alerts in the notification system. In addition, if the engineering team needs to repeatedly attend to notification issues, they might miss valuable product priorities like adding new features, improving user experience, and creating integrations.

To build a good notification system, you need to know how to measure its reliability. Read on below.

Measuring the Reliability of a Notification System

Site reliability engineering is a way to manage the operation of large software systems. The main tools in site reliability engineering are SLIs (Service-Level Indicators), SLOs (Service-Level Objectives), and SLAs (Service-Level Agreements). These are standards that form agreements between users and service providers, which specify the details of how a product is offered and the consequences if certain provisions are not met.

The key component of any reliability measurement is the way your customers perceive your product. What levels of latency in your API do your users associate with an application that's running smoothly? How long would a customer wait for the user interface to load before deciding that it's broken? How soon should asynchronous jobs complete so that your customers can proceed with their day? SLAs, SLOs, and SLIs are tools to represent numeric answers to questions like these.

An SLI is a metric that establishes the standards by which a service is to be provided to the user. A service-level indicator could be the speed of a database operation, or the size of a notification queue. These are the actual metrics that you would view in a tool like AWS CloudWatch or Datadog. The SLI, as the measurement, is what sets the basis for an SLO.

A service-level objective is the summary goal that you as a provider want to attain. An example would be a specific latency of a notification endpoint, including the latencies of underlying middleware, queues, or databases. Here, you'll especially need to understand which metrics actually matter to the customer and tailor your product objectives in that direction.

The final layer is the SLA. Your service-level agreement is a legally-binding contract with your users. It is based on the SLO and the metrics provided in the SLIs. SLAs typically reflect the targets defined in the SLO layer. An example would be an endpoint being available and returning within 1 second for 99.9% of the time. If your product falls behind the target, a customer might get the right to request a refund for your service. So SLAs tie service objectives to direct financial losses when objectives aren't met.

These components all work together to provide a specific range of metrics within which your product is operating correctly. Paying close attention to SLIs and SLOs, which should be tailored to the customer, can help identify problems before your customers do. Things will go wrong, but how you respond to each situation will make a big difference.

Notifications will be a fundamental part of your functionality. An example of an SLI could be the size of the notification queue, and an SLO could be the latency of processing a notification from creation until it's sent to the user.

While most companies will not cover their notifications under an SLA, it still might be necessary in certain circumstances. For example, a B2B CRM application where notifications need to be used as reminders of upcoming client calls will probably include notification-related standards as part of an SLA. If your product requires coverage of notifications under an SLA, take care to ensure that your product metrics and objectives are aligned with your agreements to avoid overpromising and consequent legal issues.

The idea of having to use a provider API like Mailgun or SendGrid for sending emails, or interacting with a push notification service like Firebase Cloud Messaging for iOS and Android notifications, can be a reliability concern. If you are on the fence about how using a third-party provider would impact your reliability metrics, read on below.

Is Using Third-party Provider APIs a Reliability Concern?

In considering the scope of building a notification solution, you might feel reluctant to add provider APIs into the mix and therefore focus on managing all notifications in house. Instead of using a provider like SendGrid or Twilio, you might be considering setting up email or SMS infrastructure in-house.

But is using provider APIs a reliability concern?

Courier, for example, is an HTTP API. It is true that HTTP requests can fail due to connectivity issues, SSL errors, or unexpected delays. Perhaps the customer doesn't receive an HTTP response to their API request at all. You can attempt to make such failures less common by only relying on services that reside within your network space, but due to the complexity of today's networks eliminating such issues completely is not going to be possible.
In our experience, the answer is not to avoid using APIs altogether but in how to create mechanisms to attenuate API request failures.

At Courier we built mechanisms to avoid many HTTP API issues. For example, Courier uses idempotency keys to safely retry messages without duplicate sends to the customer. Integrating idempotency and other fault-tolerant processes is a vital part of building a reliable notification system.

Now that we covered the core concepts, let's discuss specific suggestions for building a scalable and reliable notification system for AWS users.

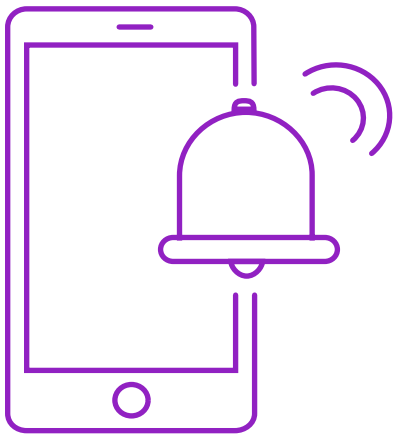
Tips for Building a Notification System on AWS

If you're using AWS, there are many tools to help you build a scalable notification system. DynamoDB and AWS Lambda are some of the AWS services that we use at Courier, and applications built using these services can be easily scalable and cost-effective to run, while requiring little to no upkeep.

Still, you should take care to avoid performance bottlenecks even when using services like Lambda and DynamoDB. Below we'll share some tips based on our experience using AWS services.

Suggestions for Scalability with DynamoDB

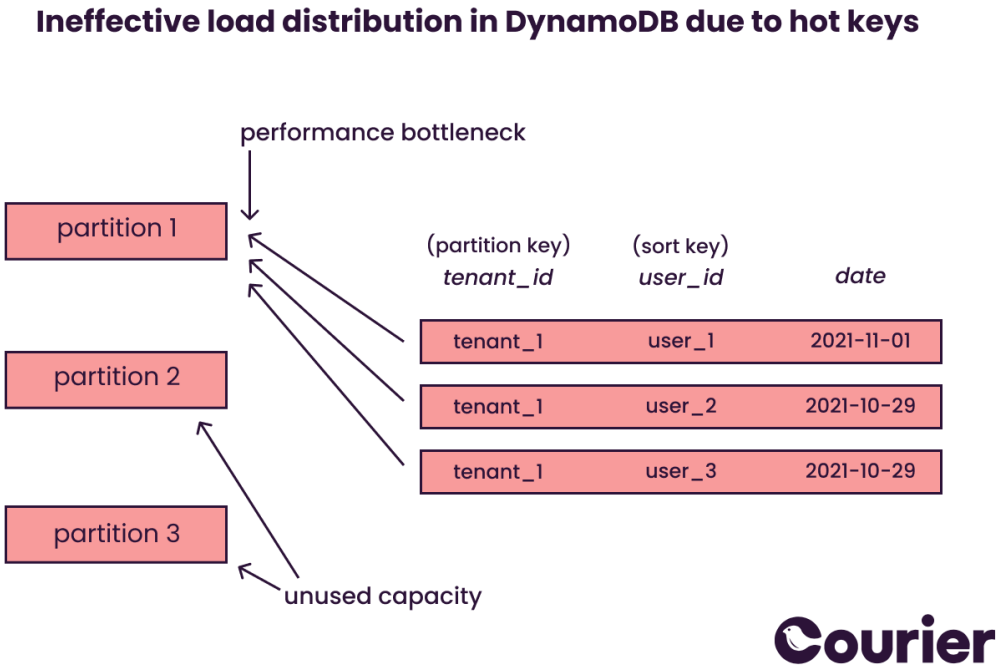
How you build for scalability depends on the tools you choose, at least in terms of how they access your data. A system is scalable when it can still perform within its service level objectives even with an increase in volume. Whichever tools you decide to use, plan for a notification system that can handle sudden and drastic increases in data volume.



When creating your own notifications application, it's crucial to pay attention to access design patterns from the outset. You'll need to understand how you'll be accessing data before you start building the application. It might not be complicated to build a simple notifications application into your product, but problems don't typically become apparent until later in the implementation or as you're trying to scale.

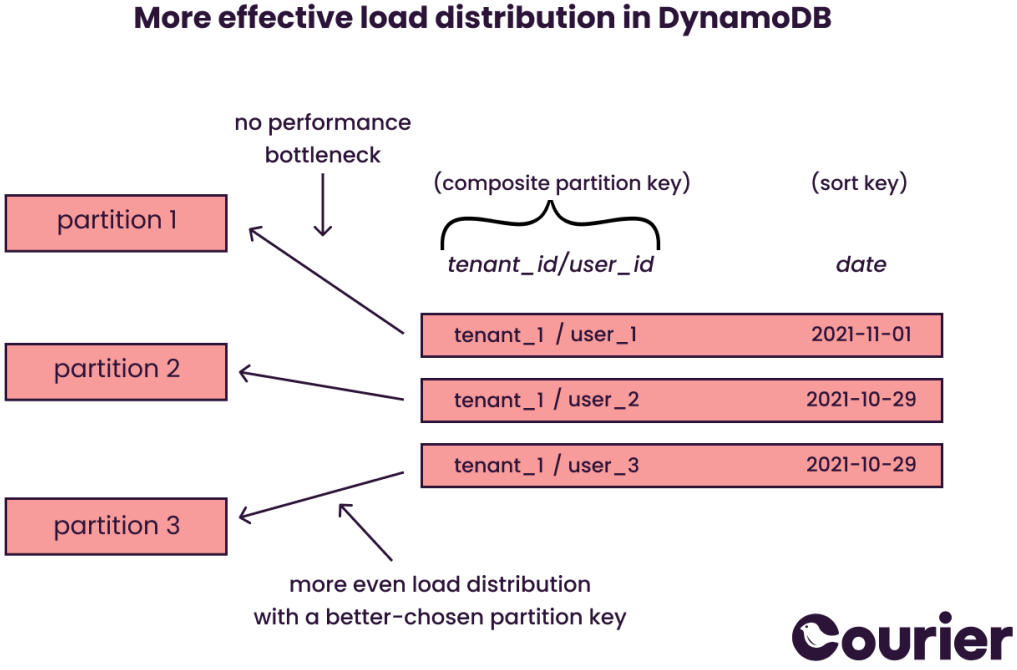
If you're using DynamoDB, a common problem with access patterns is partition key structure. DynamoDB uses primary keys that consist of two components: a partition key and a sort key. DynamoDB uses the partition key of a table to distribute the table's data across partitions. The more evenly the table's records are distributed, the higher the overall throughput of the table will be.

To determine the partition a record needs to be written to, DynamoDB runs its hashing function on the record's partition key. Based on the hashing function's output, an item is mapped to a specific physical location in the DynamoDB system. Each DynamoDB partition has a limited amount of throughput capacity. If one of your table's underlying partitions were to receive more reads or writes than your other partitions, the throughput of your DynamoDB table would be lower than if the load were evenly distributed. Overloading one partition while underloading the others due to too many records having the same partition key is usually referred to as the **hot key problem**.



Ineffective load distribution between partitions in DynamoDB.

The partitions are managed by DynamoDB itself, so the only way for a developer to address an issue with record distribution between partitions is to change the structure of the partition key. A common solution to the hot key problem is to **create a composite partition key.** In our example above, the *tenant_id* column is used as a partition key, and this configuration causes a performance bottleneck on Partition 1 when working with records for the tenant *tenant_1*. To address the problem, we can create a composite partition key by combining the *tenant_id* and *user_id* attributes. See the impact of this change in the following illustration.



More effective load distribution between partitions through the use of composite partition keys.

More effective load distribution between partitions through the use of composite partition keys.
In this example, the load is distributed more evenly because the records now have different partition keys.

Similarly, if you'll be using **AWS S3** to store attachments you send with your notifications, pay attention to your design access patterns. Improperly designed S3 bucket and key structure can cause throttling and therefore impact the performance of your application.

Depending on the volume and predictability of your usage, **reserved capacity provisioning** will be much cheaper than autoscaling or statistically provisioned capacity. DynamoDB's auto scaling feature can handle unpredictable load patterns without human intervention, but autoscaling can get expensive. If you have a predictable volume of notifications, then maintaining infrastructure to service that volume will typically cost much less than having to handle unpredictable spikes. It's even possible to mix auto-scaling with reserved capacity (see this example of cost optimization with DynamoDB).

Finally, you need to be able to monitor and analyze your performance metrics and general infrastructure. This is especially important for scalability since the metrics serve as indicators that can pinpoint issues or inefficiencies in your access design patterns. A good monitoring setup can also assist in ensuring security measures and legal compliance. For this, Courier uses Datadog, which can monitor servers, databases, and other tools.

As you're well aware, a scalable set-up for a notifications application requires serious planning before building. Since your needs will be different depending on your usage, a scalable system will have a good foundation that can accommodate higher volume without huge expense or re-building. Aim to understand your own design patterns and tools and how they can work for your application instead of against it. For example, DynamoDB is not a relational database and should not be used as such. You'll need to design meticulously early on in the process since getting it right the first time will be invaluable to your company.

Fine-tuning for Reliability

At Courier, we use AWS Lambda to run most of our notification-related code. If you're going to be using AWS Lambda for your notifications application, it's crucial to tune Lambda configuration to your required usage.

For example, we recommend modifying default timeout values. The default timeout setting can differ significantly between various AWS services and AWS SDK programming languages. In the Node.js SDK, the timeout is 2 minutes, while it's 60 seconds in the Python SDK and 50 seconds in the Java SDK.

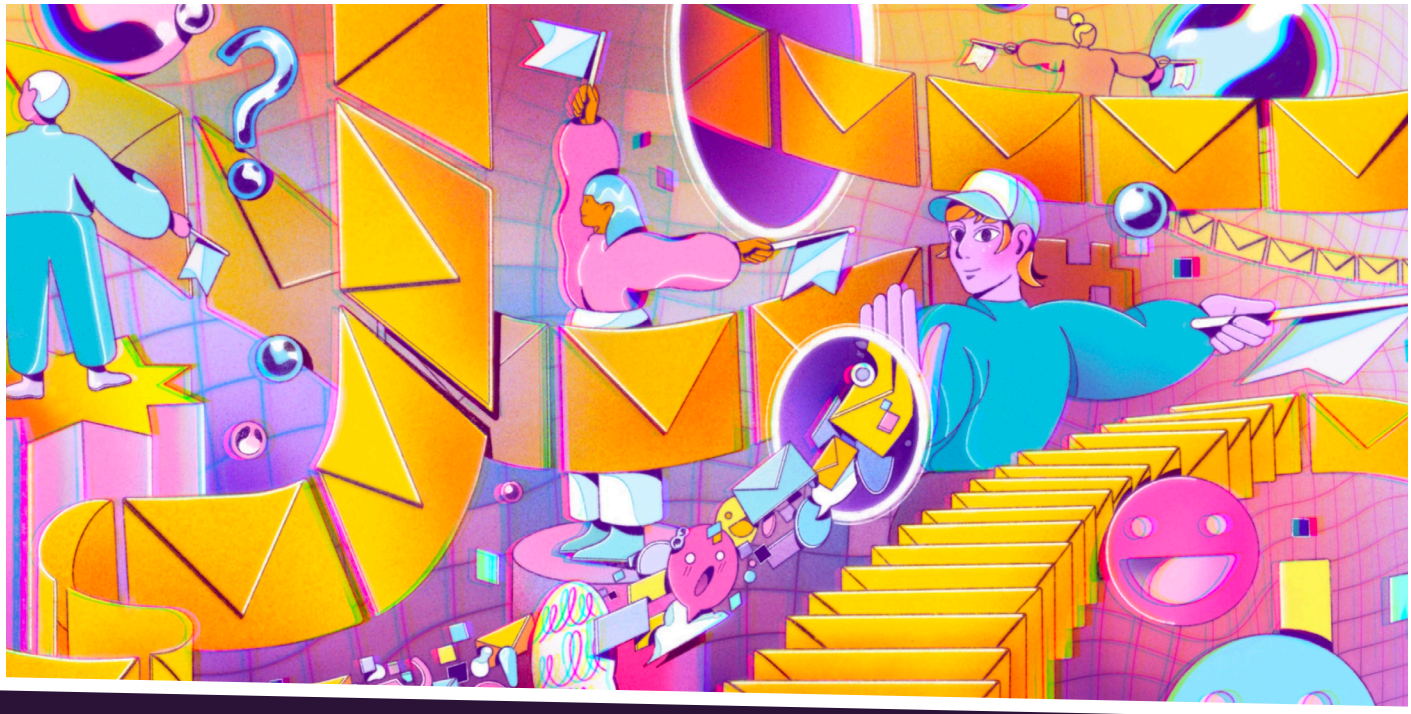
Incorrectly matching timeout settings to your use case can lead to unexpected behavior. If your Lambda function takes longer to run than the timeout configured in the SDK, you might run into unexpected timeouts.

Our typical strategy is to right-size the timeout settings between Lambda functions, the AWS SDK, and other locations in your systems where timeouts can occur. The right timeout values will depend on your needs and the ecosystem you're working with.

In addition, tuning your AWS service configurations and the AWS SDK parameters based on factors like queue visibility, numbers of retries, and polling frequency can generate a significant reliability payoff if you line the settings up in complement to each other.

Moving Forward

Building a notification system into a product is not for everyone. The process is time-consuming, complex, and expensive. Your particular requirements will ultimately dictate a preference for either functionality or cost. A startup with a product that hasn't yet found its product-market fit has to focus on finding early customers and getting their feedback. But established companies with a proven customer base will have concerns related to higher volumes, stability, and compliance. This would require more functionality and higher maintenance costs. Scaling reliably can be hard, but despite the complexities, it can be done without sacrificing throughput for maximum reliability.



Routing and Preferences

Notifications serve a range of purposes, from delivering news to providing crucial security alerts that require immediate attention. A reliable notification system both enables valuable interactions between an organization and its customers and prospects and also drives user engagement. These systems combine software engineering with the art of marketing to the right people at the right time by focusing on routing and preferences.

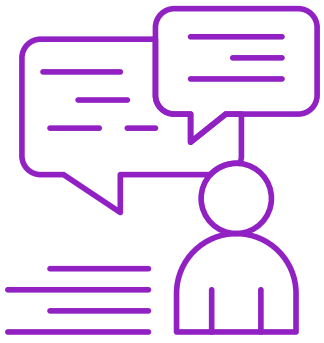
Building a service capable of dynamically routing notifications and managing preferences is vital to any notification system. But if you've never built a system like this, it might be difficult to figure out what the requirements are and where the edge cases lie.

In this section, you'll learn invaluable points to consider when building your own routing service. You'll understand the requirements for multi-channel support and in choosing the right API providers. You'll also learn how to design user preferences so that you can make the most out of each message.

Multi-channel Support: A Necessity

Let's say that you have just built a web-based application. The first channel that you'll use to connect with your users is likely email because of how ubiquitous it is. However, with the diversification of channels and depending on your use case, email might not be the most efficient notification channel for you. Compared to other channels, emails typically have a low delivery rate, a low open rate, and a high time to open rate. It's not uncommon for people to take a full day to even notice your email. If your email gets to the user, it might take awhile before they open it, if at all.

To engage with your users more effectively, you'll want to support channels across a broad range of systems not limited to any one application or device. It's vital to understand not only which channels are most relevant for you but also for your users. If you opt to use Telegram and your users don't have it, it won't be a very useful channel to interact with them. Multi-channel support is also vital because while you might pick appropriate channels today, you won't know which channels you will need to support in the future. Typically, the more appropriate channels you support, the higher the chances of intersecting with applications your users actually use now and in the future.



Choosing Notification Channels and Providers

You'll have to select relevant channels and appropriate providers for each channel. For example, two core providers for mobile push notifications are Apple Push Notification Service (APNs) and Firebase Cloud Messaging (FCM). APNs only supports Apple devices while Firebase supports both Android and iOS as well as Chrome web apps.

In the world of email providers, SendGrid, Mailgun, and Postmark are all popular but there are hundreds more. All email APIs differ in what they offer, both in supported functionality and API flexibility. Some providers, like Mailgun, only support transactional emails triggered by user activity.

Other providers, like SendGrid and Sendinblue, offer both transactional and marketing emails. If your company opts for a provider that can handle both, you'll still want to separate the traffic sources, by using different email addresses or domains, to aid email deliverability. If you only have one domain for sending both types of emails and the domain gets flagged as spam, your critical transactional emails will also be affected. Whichever provider you choose, you'll still want to meticulously verify your DKIM, SPF, and DMARC checks, and domain and IP blacklisting using your own tools or a site like Mail-Tester.

Making requests and receiving responses also differs with each email API provider. Some providers, like Amazon SES, require the user to handle sending attachments, while others, like Mailgun, provide fields in the API schema for including attachment files directly.

There are minute variances in formatting HTTPS requests. The maximum payload sizes range from 10MB with Amazon SES API and up to 50MB with Postmark. There are also differences between the rate limits for requests.

In terms of API responses, Amazon SES provides a message identifier when an email is sent successfully through the API, but, for example, SendGrid returns an empty response in that situation. The HTTP response codes also differ slightly depending on the provider. For example, AWS SES uses the response code 200 for successful email send operations, while Sendinblue uses 201, and SendGrid uses 202.

No matter which provider you end up choosing, don't build your application solely to fit their logic and specifications. If you do so, it will be much more difficult to change providers in the future as you'll have to overhaul your backend. It's crucial to invest in a layer of abstraction based on your own paradigm.

Dynamically Routing Notifications Between Channels

How do you determine which channels to use and when? Just because you're able to use email, SMS and mobile push doesn't mean that you should use all of them simultaneously, since doing so carries a high risk of annoying your users. This is where you begin to formulate an algorithm to route messages between the different channels and the different providers within each channel. The algorithm needs to be robust to handle delivery failures and other errors. For example, if the user hasn't engaged with a push notification after a day, do you resend it or use email instead?

You can begin constructing the algorithm using basic criteria. For example, if there is no phone number, eliminate SMS as an option for that user. If email is the primary channel, opting to send at 10 a.m. or 1 p.m. local time typically improves read rates. If the user is present or active in the app, consider sending an in-app push notification instead of an email. Finally, and especially important, get your user's preferences for how and when they want to be contacted and integrate these preferences into your routing service.

Adding User Preferences to Your System

Once you've got your channels, providers, and routing algorithm figured out, you need to think about providing users with granular control over notification preferences instead of just a binary opt-in/opt-out switch.

Consider this: if you only allow opting in to or out of all notifications at once, your users might unsubscribe from all your communications because they find one specific notification annoying. As a result, you will lose out on valuable user engagement.

With granular control over preferences, a user identifies exactly how and when they hear from you. If a user doesn't like email but wants SMS messages (not common, but possible!), they can adjust their preferences and keep the SMS line of communication open. Every enabled notification channel is another opportunity to engage the user in a way that's productive for them. From the end user's perspective, it's empowering to control how and when they are contacted.

Note that for some channels, the user's preferences should be ignored. For instance, two-factor authentication should go to SMS or mobile push regardless of the user's preference for email. The

possibility to override the default logic should be incorporated into your algorithm while you are designing your routing engine.

If you want to take user engagement further, allow users to opt-in/opt-out of specific channels, frequency, timing and topics. You can allow them to set up their preferences based on time of day, frequency per period, or to specify more than one email address. You can give them the option to receive transactional, digest emails, daily newsletters, or only the critical ones. You can also allow them to redirect their notifications to another address, for example if the user is out of office.

Granular preferences also extend past the dominion of PMs and the user's experience. Granularity of consent is becoming part of privacy compliance laws in Europe and in the state of California and might follow elsewhere in the future. Separately, granular preferences are an extremely advantageous analytical tool for the marketing team to improve brand strategy and personalization efforts. Is there a particular channel or topic that seems to be more popular? That information can be highly helpful to pivot in line with your users and grow your company.

Tips for Future-proof Maintenance

When you're starting with notifications for a new product, there is nothing wrong with sticking to one channel and one provider. The most important principle to keep in mind is to design your notification system so that you can expand it in the future. You should leave the door open to include more providers when you need them.

Don't assume that **API paradigms** are the same for each provider or notification type. For example, you want to send an email, and if delivery fails to send a push notification instead. But you won't get a 400 HTTP response from the email provider in case of failure. The provider will retry your email over a couple of days. Instead, you'll want to include webhooks or queues to notify you of the failure, and you'll need to track the state of the message here. If you make blanket assumptions about how API calls work or how errors are returned, you'll have trouble adapting to a different paradigm in the future. Instead, you can add a layer of abstraction on top of the API.

It's also invaluable to **centralize the way you call the provider APIs**. If you spread out calls to an API throughout your code base, it will be more difficult to integrate other channels or API providers in the future. Let's say you're starting with email and AWS SES as the provider. In two years' time, you might decide to integrate mobile push notifications as well. What might that look like? The incurred technical debt will include scouring the code base for all instances of calls to the AWS SES API before you can integrate mobile push as an additional channel. But with centralized calls, you'll have more consistent, cleaner, and reusable code as you grow.

How Many Notification Channels Should You Have?

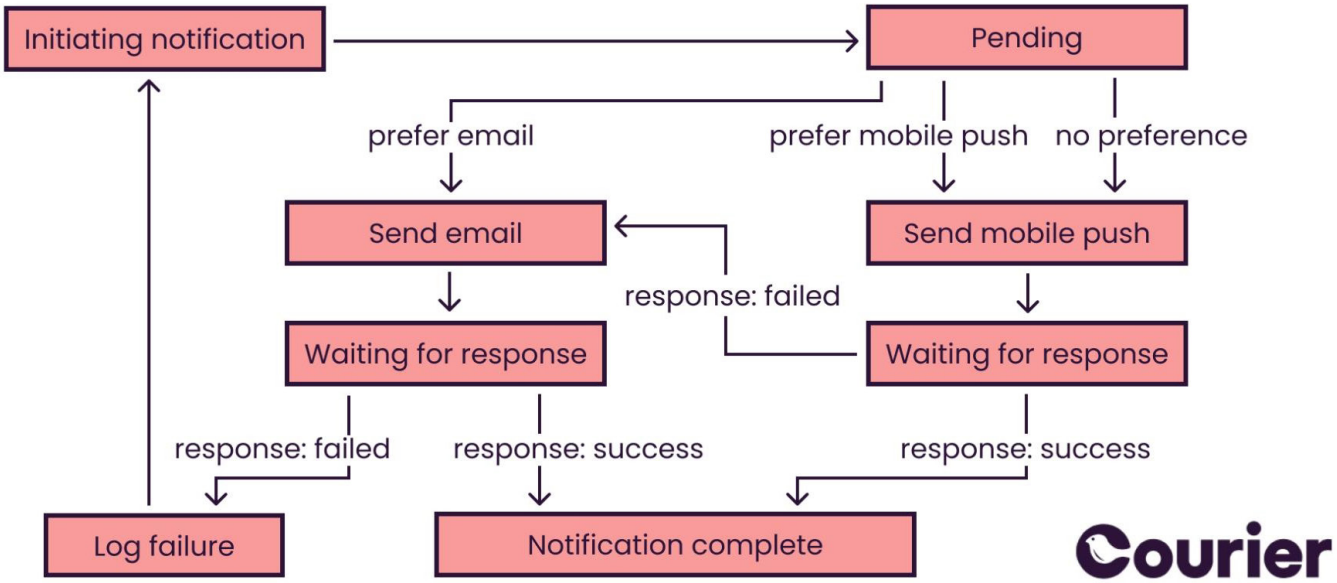
Typically, having three or four channels that are relevant to your product is an ideal scenario for a mature product. When you intersect channels with the preferences and availability of users, you create higher levels of complexity for your algorithm. Offering many channels for notifications might become too complex to maintain. But offering too few channels might harm your chances of interacting with users since some channels might not be viable for all users. For instance, you might decide to offer email and push notifications. But if a user didn't download your product, your interaction with them is limited only to email.

Best Technologies for Routing and Preferences Engines

It ultimately pays to choose technologies that will be a good fit for your routing and preferences needs. There will be a great deal of asynchronous programming, as the routing service will often be waiting to receive responses for each function. You'll want to pick a language or a framework that allows you to respond to async events at scale.

The routing service also involves considerable state tracking, as most of the routing will depend on waiting on a response for each notification before changing state. The routing service will also need to be re-activated every time it receives a response from a provider and will need to determine if the notification was sent successfully or if it has to pursue next steps. See the example below of how a notification function's state might be tracked.

An example of notification state tracking



At Courier, we use AWS Lambda. Since our usage tends to come in bursts, serverless technology allows us to adjust and scale for changes in demand throughout each day as well as handle asynchronous operations efficiently.

Don't Forget: Compliance in Notification Routing

When creating your own routing and preferences service, you will need to ensure that whichever channels you implement are fully compliant with applicable laws. For example, there are legal mandates on how users may be contacted or how they can unsubscribe from contact.

For commercial email messages, the CAN-SPAM Act of 2003 is a federal United States law that spells out distinct rules and gives recipients a way to stop all contact. Penalties can cost as much as $16,000 per email in violation. This law also outlines requirements such as not using misleading header information or subject lines, identifying ads, and telling recipients how they can opt out of all future email from you. The opt-out process itself is strictly regulated.

For SMS, the United States Telephone Consumer Protection Act (TCPA) of 1991 sets forth rules against telemarketing and SMS marketing. Under this law, businesses cannot send messages to a recipient without their consent. This consent needs to be explicit and documented. The consent is also twofold: recipients need to consent to receiving SMS marketing messages and they need to consent to receiving them on their mobile device. Recipients need to be provided a description of what they are subscribing to, how many messages they should expect, a link to the terms and conditions of the privacy policy, and instructions on how to opt-out.

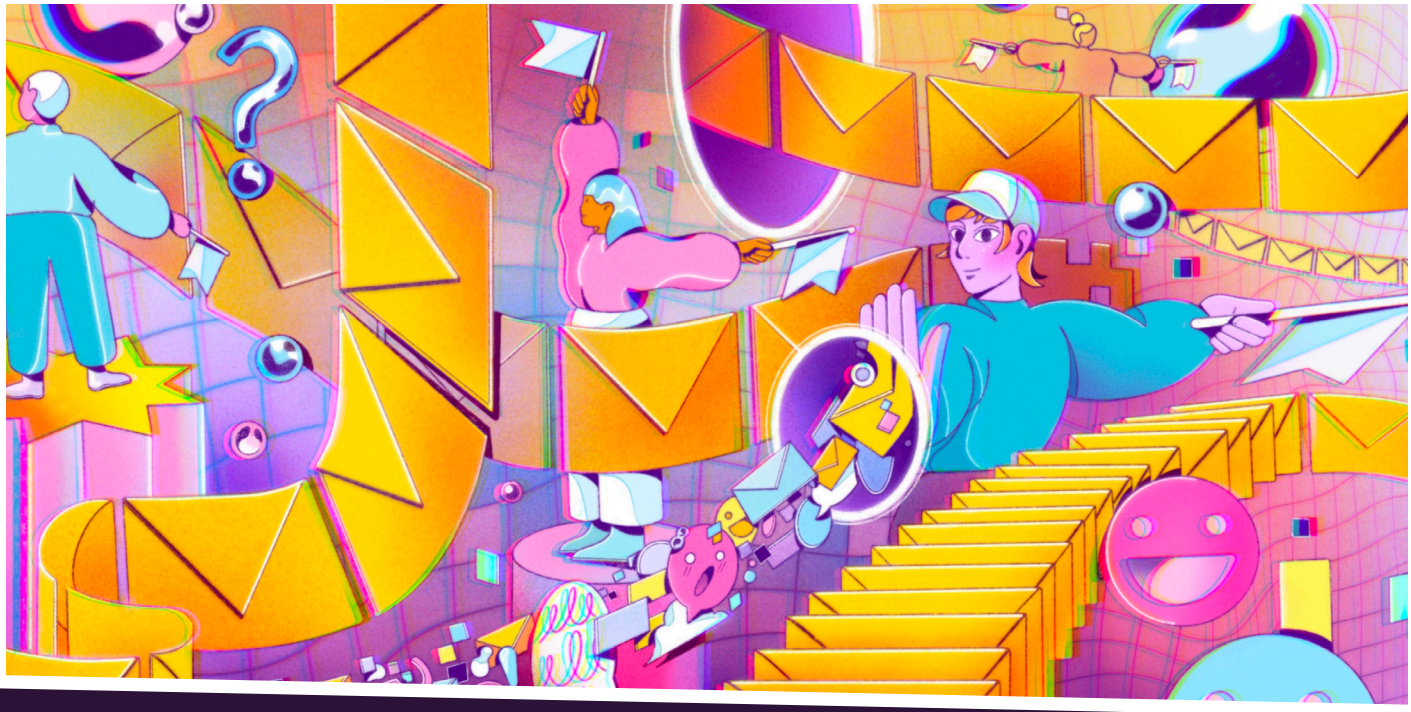
In California especially, the California Consumer Privacy Act (CCPA) of 2018 provides additional rights for California residents only. These rights include the right to know which information a company has collected about them and how it's used as well as the right to delete it or to opt-out of the sale of this information. Information that qualifies under the consumers' right-to-know includes names, email addresses, products purchased, browsing history, geolocation information, fingerprints, and anything else that can be used to infer preferences. Should a consumer request this information, the company has to share the preceding 12 months of records, and also include sources of this information and with whom it was shared and why. In 2020, California Privacy Rights Act (CPRA) of 2020 amended the CCPA. The CRPA provides further consumer rights to limit the use and disclosure of their personal information.

Other countries have their own compliance laws for businesses reaching out to leads and customers. Canada has its Anti-Spam Legislation (CASL). The European Union has the General Data Protection Regulation (GDPR) which now also covers granularity of consent. The United Kingdom has its own regulations along with the GDPR, the Privacy and Electronic Communications Regulations (PECR) and Data Protection Act.

Compliance itself needs to be integrated at the development level. Providers, like SendGrid, don't know what you're sending. It's up to the infrastructure to ensure that all applicable compliance laws are followed for their choice of channels.

Moving Forward

The level of notification customizability and routing options you decide to implement will ultimately dictate a preference for either maximizing user engagement or optimizing cost. This section taught us about the necessity of sending data for notifications to the right people, at the right frequency, at the right time and how this can be done through routing and customized preferences.



Observability and Analytics

Developing an application can often feel like you're building in the dark. Even after development, gathering and organizing performance data is invaluable for ongoing maintenance. This is where observability comes in—it's the ability to monitor your application's operation and understand what it's doing. With close monitoring,

Observability is a superpower that allows PMs to use various data points to foresee potential errors or outages and make informed decisions to prevent these from occurring. As you build your product, consider the implications of having complete observability built into your notification system. As a PM, you'll need to identify and quickly resolve issues by understanding how your product is performing. In the bigger picture, observability ties your technological infrastructure to your overarching product and business objectives. These key insights will also help to scale the product and manage data as your business grows.

Observability Use Cases

You're here because you want to build an application with a powerful notification system that can rival those of existing products. In this guide, you'll learn why observability and building strong monitoring mechanisms are crucial. Here are four core observability use cases.

1. Track and use logs for debugging
2. Improve customer service and experience
3. Holistic view of your product
4. Analytics for business development

Logs for Debugging

Telemetry logs are the backbone of an observability system. The more infrastructure you have, the more data there will be from each instrument and service. You need to be able to understand this data. Logs can provide additional context that allows PMs to determine where or why certain issues might be occurring, and effectively, how to fix them. For example, if every API request to your notification system results in a specifically formatted log line, it becomes possible to scan those log lines for anomalies. Event logs of particular actions occurring in the system, like privileged access or settings changes, can sometimes shed light on unpredicted behaviors in the system. At the very least, you should have some kind of safety net or global catch to notify you when errors creep up.

When users are unable to receive notifications, you can use various logs to determine what factor(s) prevented that notification from going through. If, for example, your app is unable to deliver messages to 20% of your users, logs would reveal that those same 20% of users are using your application on a specific device type. You'll know right away that your application has a bug that prevents it from functioning properly on that device type and act accordingly. You can also update your system to prevent this issue from occurring for future users.

Customer Service

Let's say a user has contacted you to report that they have unsubscribed from emails but is still receiving them. Your customer care team should be able to note relevant logs or errors, communicate with the user to ensure a good relationship, and also feed that information to the development team for potential resolution.

Observability can also help you improve overall user experience, present and future. As you consider important data points to observe within your system, think about what might impact your product, specifically through the lens of your customers. If you can see some metric that might impact user experience, work to improve it before it's an issue. For example, do your time-sensitive notifications get delivered as quickly as they should? Are all messages being delivered only once? If you're using multiple channels, are messages being routed correctly?

Additionally, if you're using a third-party provider like SendGrid or AWS SES, you should absolutely observe connection health. If there's a provider issue, you could notify your customers, such as through a status page, that notifications might not be working optimally. You might not be able to control the operational status of your providers, but you can still take action to maintain your customers' trust.

Holistic View of the Product

A proper observability environment should provide you with a holistic perspective of your application's state. Based on usage and performance, you can clearly reason about how certain factors might affect your product. You can gather a real understanding of the rhythm of your notification system. You might see that your users are receiving fewer notifications at specific times in the day or year. How do you know if this is due to your application sending fewer notifications, or because they're not getting delivered at all? With notifications, data can fluctuate drastically. Peaks in error rates can be cause for concern and require immediate engineering support, while fluctuations in send volume can be observed with caution to ensure that the application is sending the right amount of notifications.

As you set up your observability, try to expose the data through user-friendly dashboards and interfaces, such as those that Datadog and Honeycomb offer. All of the collected data should be organized to provide clear insights on the application's behaviors. Proper data visualizations are invaluable and should be tailored to more than just product teams. For the customer service team, it is helpful to understand a specific user's experience when something goes wrong. Likewise, commercial or marketing teams can glean insights for business development.

Analytics and Business Development

If your users trust that you can monitor your application efficiently and resolve issues quickly or even use data to prevent their incidence, you'll build a strong foundation and customer base. How you organize your observability system should ideally connect to your service-level metrics, which should be recorded in your SLAs (service-level agreements) that you have with your users. The observability data will be more relevant to your business if it is connected to your SLIs (service-level indicators) and SLOs (service-level objectives). An observability system that monitors all varieties of resource consumption without this connection to user experience might not foster the type of growth you want.

Tracking and analyzing data on how your users are interacting with your notifications can help drive business development opportunities. Link tracking, for one, is a core observability component within a notification system. Did the user click on your notification? When and how many times?

Analyzing your observability data, and especially what you do with the resulting insights, is vital to further business growth. Your observability metrics will allow you to determine if the product is meeting business expectations. For instance, you can use observability data to make scaling decisions. If you want to scale, how might increases in volume affect your system? As you aim to understand the signals from the noise in all of your observable data, every signal you find needs to drive meaningful change. This is where you will find opportunities for further development.

Making Your Notification System Observable

Once you know how to use data to effectively monitor your application and make informed decisions, where do you start? The ultimate goal is to design your observability environment so that it is able to understand data, compare the data between various channels and infrastructure, and make it actionable.

There's a way to structure data to make it more useful to engineers, customer service, and business development teams. Making sense of this data requires two key measures: the *correlating and normalizing events data*. Correlation illustrates how different events are connected to each other and how they are connected to different users. In a notification system, this means that every outbound message, the receipt and opening of a message, and every click on the notification are measured in the ways they are statistically related to one another.

Normalization refers to how we understand data points from different sources or channels and record them in a way that makes them comparable. That means re-scaling the data so that it all varies on a *similar scale*. For example, how would data from email notifications from SendGrid compare to SMS notification data from Twilio? These are not only different companies, but also entirely different channels.

Correlation and normalization of data allow you to have a more complete understanding of how your product's notifications are performing, both in general and in relation to one another. You would then be able to filter through data and for example, find all events related to one user. This would include all email, SMS, direct message, and push notifications regardless of how and where the user received them.

You can also filter through the data to find all emails sent out on a specific date under certain conditions, like for multiple users or in specific regions. Ultimately, if you're working with several providers for your notification system, correlating and normalizing data points will be a vital step to achieving observability.

Observability Tips for Serverless Notification Systems

Your setup for an observability system will depend on your tech stack. For the sake of this example of a serverless notification system, we'll use AWS DynamoDB and Lambda. Generally, observability boils down to metrics, logs, and traces—and how you manage them.

Since we're using AWS DynamoDB and Lambda, AWS CloudWatch is a great tool that provides built-in monitoring in connection with many other AWS services. CloudWatch collects both custom logs and those of other AWS services, as well as infrastructure metrics.

At Courier we import these metrics and logs into Datadog, which aggregates everything on a dashboard. Datadog can be integrated with both DynamoDB and Lambda, as well as hundreds of other services.

In a notification system using Lambda and DynamoDB, you should monitor all default performance metrics such as the number of functions getting called, the number of rows being modified, and so forth. Some of the notification-specific metrics not already mentioned here include latency, error rates, and request rates.

For DynamoDB in particular, it is valuable to monitor access patterns, such as inputs and outputs, in order to avoid hot keys. A tool like CloudWatch Contributor Insights can help identify and analyze these access patterns.

AWS Lambda can automatically capture logs and then connect them to AWS CloudWatch with the Lambda Logs API. Through this API, extensions can subscribe to function logs, extension logs, and the Lambda platform logs for events and errors. Datadog can import these as well.

For logging DynamoDB activity, AWS offers CloudTrail. All API calls for DynamoDB are captured as events. You can search through the event history, or you can create a trail for ongoing event delivery to an AWS S3 bucket. This allows for an extended record of events and you can integrate these logs to CloudWatch.

If you're using a tool like Datadog, you might forward your CloudWatch logs and metrics to Datadog using a Forwarder Lambda function. If you're also using Kinesis in your tech stack to quickly process streaming data, you can use their Firehose delivery stream to forward logs to Datadog as well.

Important Logs to Track

There are many vital metrics and logs to observe in a notification system. Some, already mentioned in this article, include the sending of notifications, receipt and opening of notifications, and any clicks on notifications. Other important ones include deliverability, open rate, and conversion rate. You'll also want to note the channel and provider for each notification, such as Twilio for SMS. You are inherently looking for two different things. The first is how your notification system is operating so that you can resolve potential bugs and strive for improvement. The second is how successful your notifications are in engaging with your users. Any log or metric that might help you define those two components will be useful.

Finally, traces can greatly help understand the context of logs or metrics. Traces track requests from beginning to end through all of the components in your tech stack. Tracing is especially key when your tech stack involves several intertwined systems, as it can help identify bottlenecks between those systems. Traces are normally integrated through logs, such as a unique ID for each request. One idea would be to attribute each operation in DynamoDB and Lambda to a specific notification that is sent by a specific user.

Moving Forward

If you're building your own notification system, observability is vital for both maintaining and scaling your product. It is a preventative measure that can improve performance and health of your system. Whatever the relevant observability metrics may be for you, the data should be measurable, actionable, and meaningful. Remember that it's what you do with your data that impacts your business the most.

Conclusion

The purpose of this ebook is to help you, the product manager, understand the complexities and intricacies of building an in-house product notification system and get you moving in the right direction. Your CTO assigned an enormous project to you, one that large companies like Uber, Slack, and LinkedIn have entire teams for, but by following these steps and digging into the details, you can get going on this project on your own.

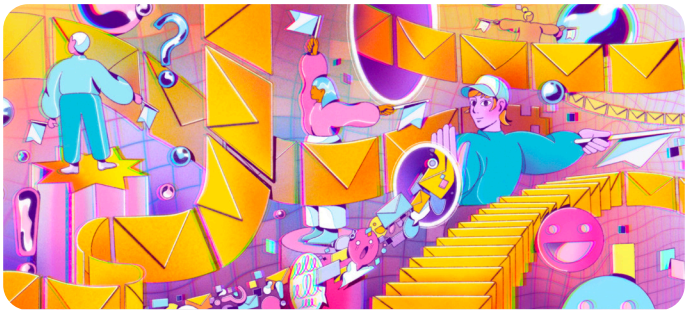
Visit http://courier.com to learn more about how you can outsource this project entirely to a third party and check our blog and other content for more details on how to do it yourself.

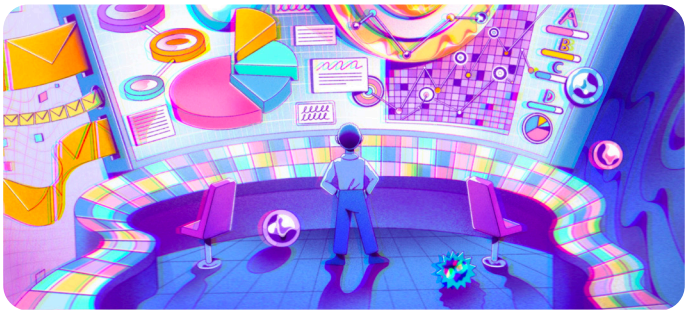
Artist's Statement

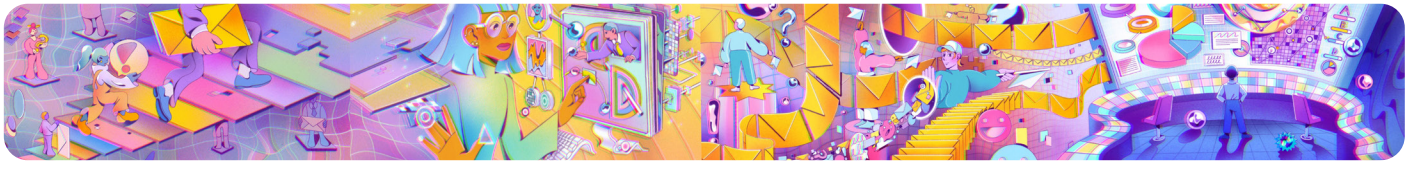
When I was approached to illustrate The PM's Guide to Building Product Notification Systems, I immediately knew I wanted the imagery to link together and create a grand narrative that sort of mythologizes the design process that's being described. As someone who's very drawn to conceptual depictions of cyberspace and digital worlds, making sure that the Courier Dream Land felt vast but still full and teeming with warm human presence became the main priority.

Rebekka Dunlap

https://rrebekkaa.com/








Courier
www.courier.com