Meet Barry, IIJ's Tool for Rapid Fault Resolution

3.1 Background to Barry's Deployment

To provide stable, high-quality services, IIJ must attend to a range of operational tasks. Key among them is troubleshooting, which involves restoring service-providing systems when they are unable to maintain a normal operating state because of hardware or software faults. IIJ uses an internally developed operations system called Barry for troubleshooting. Here, I describe how Barry works and what it does.

First, however, I would like to talk about how we dealt with faults before Barry. Normally, the services we provide are constantly monitored for any anomalies in the equipment and functionality provided. When service anomalies arise, an alert is generated, prompting us to take action. The first step is to find troubleshooters capable of dealing with the issue. At IIJ, we call this escalation. Next, the troubleshooters begin the job of restoring the service to a normal state. The actions taken differ depending on the service, but generally the process involves the people involved communicating with each other and using various tools to investigate and record the issue as they work toward a solution.

This is how IIJ had been dealing with faults until now, but this approach had its problems. So we revised our approach and developed an operations system called Barry to facilitate smoother troubleshooting.

The two main problems with the previous approach were as follows.

■ Problem 1: Finding troubleshooters and accurately communicating the details of the issue

We need to find people quickly. We currently still use phone calls to contact candidate troubleshooters and ask if they can tackle the issue. The advantage of using the phone is that we can call them continuously. Emails and other messaging can also be used to escalate an issue, but such messages are usually only sent to candidates when the issue arises. With the phone, however, we can continue calling the person until we reach them, so we have a higher success rate in getting hold of the right people for the job. The flip side of this is that the people manually making the phone calls are tied up until the troubleshooters are found. This issue can be addressed by using automated phone calls, but this entails one-sided voice communication, so it can be difficult for the person to clarify and confirm the details, and the issue of automated calling system cost also remains. And there are limits on how many people can be called at once.

Also with phone calls, people can mishear or fail to hear what was said, and communicating English abbreviations and symbols is also difficult. An advantage with email-based escalation, however, is that these issues do not arise and it is easy to communicate complicated information.

Based on the above, we identified the ability to call people continuously and to accurately communicate information as two key points. Solving these issues should speed up the initial part of the troubleshooting process.

■ Problem 2: Reducing the load on troubleshooters

The work of dealing with faults puts a load on the troubleshooters in several ways. Systems can recover from some simple faults automatically, but the faults we are talking about here are those that require a troubleshooter with deep knowledge of the service to tailor a response to the situation. So high-level knowledge of the service and the right skills to address the issue are required. Other sources of pressure on troubleshooters include the need to sometimes respond on holidays or at night and demands for the rapid restoration of service. And on top of the high difficulty of dealing with faults, they need to communicate and share information with other concerned parties.

Under these circumstances, it was often the case that certain individuals, depending on the system, would handle much of the work. But it was difficult to ascertain how much of a skew there was in the workload. To enable troubleshooters to concentrate on dealing with the issue, we wanted to make it as easy as possible for them to perform the peripheral tasks.

3.2 Addressing the Problems

We thus looked at using an operations system to enable a fast response and reduce the load on troubleshooters. Adopting an existing tool was also an option, but none of the tools available were immediately suitable to IIJ's own response process, so with an eye to optimizing the internal workflow, we decided to develop a system in-house. While this approach requires cost outlays, it also has a strong advantage in that we can continuously improve the system as needed.

Firstly, we had the idea of building it as a smartphone app to solve the problem of getting hold of people. The concept was to mimic the incoming call screen and to display a text message once the "call" had been answered, thus realizing the advantages of both modes of communication. This simultaneously has the advantage of messaging, in that it is text-based, and the advantage of phone calls, in that people can be called continuously. It also does away with the limitations of phone calls , opening the door to the notion that we could customize the system in terms of the escalation sequence, how many people are called at the same time, and so on. We saw the potential flexibility to tailor the call to the style of the operations team.

In terms of reducing workloads, we felt we needed to be aware of what aspects of the process troubleshooters find inconvenient, so we told several operations personnel about the idea of using smartphones to page them and asked for their thoughts. Many of the responses indicated that sharing information was troublesome. The process of dealing with faults involves understanding the details of the fault, sharing information when implementing the response, and incident tracking. At the time we spoke to the operations personnel, each operations team was using different tools to share information in real time when implementing a response, including IRC (Internet Relay Chat) and SaaS communication tools. Various methods were also used to record the faults as incidents, including email and Wiki/ticket systems. Another inconvenience was knowing that a fault has occurred but being unable to tell what the response status is when away from the PC screen. Given these points, it was apparent that we needed to integrate information sharing and tracking for the people to whom issues are escalated. This is because tool ease of use has a major impact on the efficiency of the response process. Our focus with the new operations system was on usability, with the opinions of troubleshooters taken into account.

3.3 Barry's Featuresn

We started implementing the new operations system under the name Barry. The name comes from the famous Swiss mountain rescue dog and signifies our hope that the tool will come to the aid of those dealing with system faults.

Barry's features are divided into three parts: server, Web frontend, and mobile app. The server implements core functionality, such as escalation and incident tracking, and exposes it as a gRPC API. The Web frontend and mobile app use the server's API to provide a UI (Figures 1

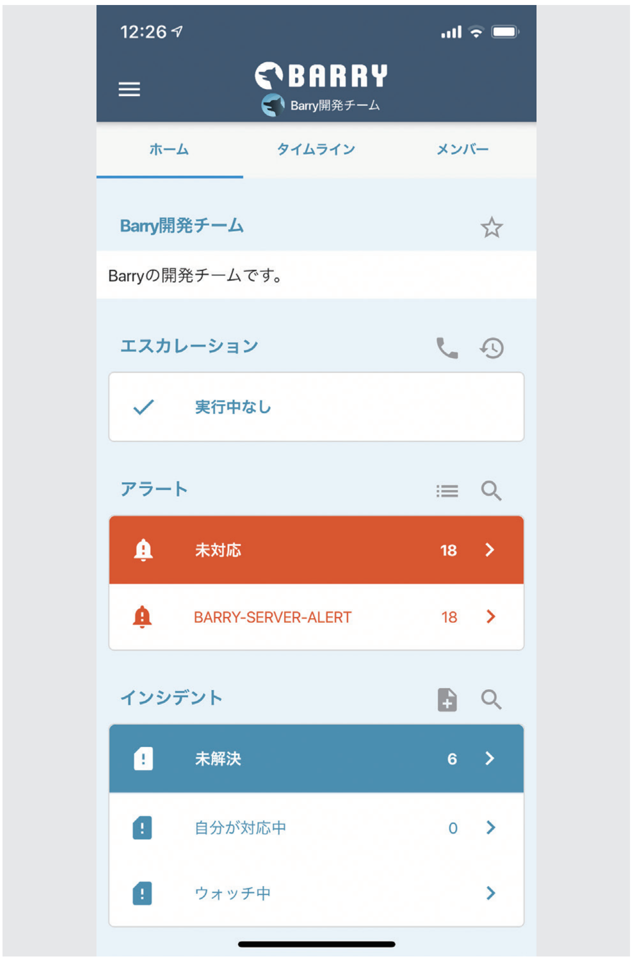


Figure 1: Barry's Mobile App Screen

& 2). Conducting the entire troubleshooting process via the mobile app would be a bit daunting at present, so we have each part doing what it does best. We see the mobile app as mainly being for calling people and facilitating simple communication, and we have structured the system on the premise that the bulk of the incident response will happen on PC via the Web frontend.

Using smartphones as a tool makes it possible for people to offer advice and other support in circumstances when previously they would not have been able to tell what was happening or be involved in the response. The mobile app is made available internally through a mechanism for distributing apps within an organization.

I will now go through specific features we implemented in Barry.

Feature 1: Flexible calling

To implement the smartphone calling feature, we used the same technology as an ordinary phone call app. When the server is given a request to initiate escalation, it sends a smartphone notification to the operations team for the service on which the fault has occurred (Figure 3). Upon receiving notification that an escalation has been initiated, the mobile app displays the incoming phone call UI. Once users answer the call, they launch the mobile app and review the details recorded on the server; they then reply via the app to say whether they can deal with the issue or not. Once an affirmative answer is received, the escalation process is complete. This is the basic mechanism, and the operations team can freely configure the server for the desired call order, number of devices called simultaneously, ring time, and number of retries.

There are also two patterns for initiating escalation: automatic and manual. An escalation can be generated automatically in response to a service monitoring alert, which I mentioned above. The system also supports manual escalation so people can be called in emergencies independent of whether an alarm is generated.

Feature 2: Integrated information tracking

An issue we identified was that dealing with faults was burdensome for troubleshooters because they were using all sorts of tools in the process. Barry provides functionality that integrates the entire process from escalation to incident tracking. In addition to the calling feature, we also implemented an incident tracking mechanism. The tool is functionally equivalent to an issue tracking system and allows people to record details of the fault and track response status.

Specifically, each failure is deemed to be an incident, and the operations team's communications with each other and updates to response status are recorded up until the issue

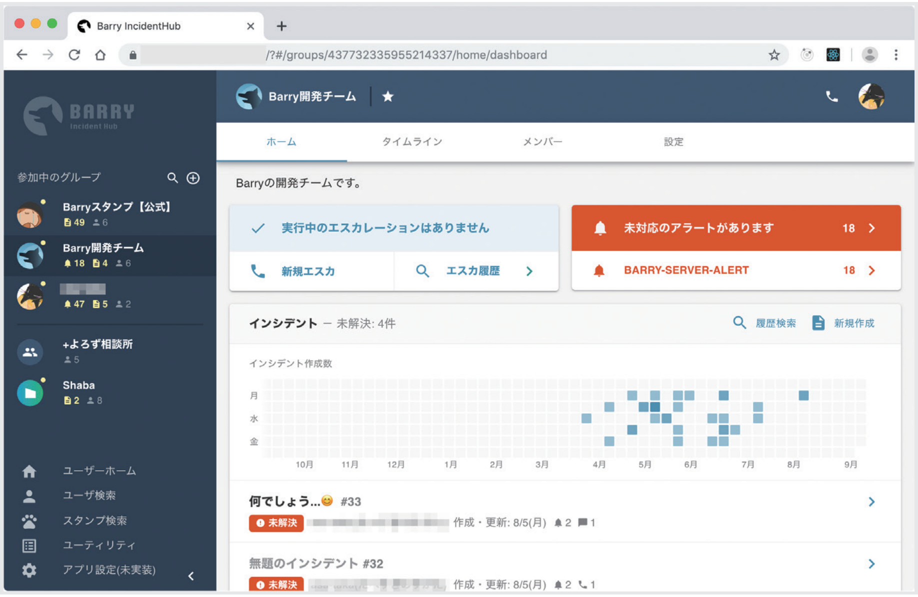


Figure 2: Barry's Web Frontend

is resolved. Alerts generated can be linked to incidents. This recording of incident histories makes it possible to refer to past examples when addressing faults.

Implementing this functionality made it possible to handle the entire process from fault occurrence through to resolution via a single tool. And because Barry supports both Web and mobile app interfaces, frontline troubleshooters can view the status and make comments even while on the move.

Ease-of-use considerations

The benefits of implementing functionality to integrate a range of tools would be limited if it ultimately resulted in lower efficiency. We therefore made ease of use a priority with Barry's tools.

We interviewed troubleshooters when designing the system, and we then created mockups and asked for feedback to ensure we were on the same page. Repeating this process several times clarified what features were needed, and it also gave us early feedback on usability. We created a lot of fine-grained functionality, so here I will run through the major features.

Activity history display

To make it easy to see how often a phenomenon occurs and how much work is involved in rectifying the fault, we implemented statistics and visualization. This feature graphs a time series of alerts and activity for each user (Figure 4). Displaying alerts on a timeline enables efficient analysis of the circumstances under which faults are occurring. And making it easy to see the operations team's activity history helps managers understand what is going on more accurately than before.

The timeline display feature shows events in order of occurrence. When using Barry, users see a lot of alerts and new incidents/comments. It's not uncommon for users to have multiple operations teams, and it can be difficult to understand what is happening when many events occur at once. The timeline feature displays events for each user in chronological order, making it easy to keep track.

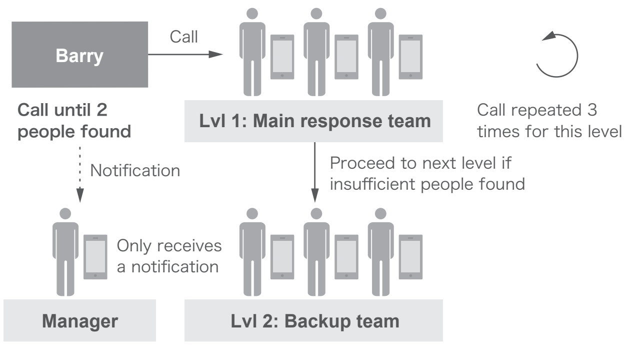

Barry
Call
Call until 2 people found
Lvl 1: Main response team
Call repeated 3 times for this level
Notification
Proceed to next level if insufficient people found
Only receives a notification
Manager
Lvl 2: Backup team

Figure 3: Barry's Calling Process

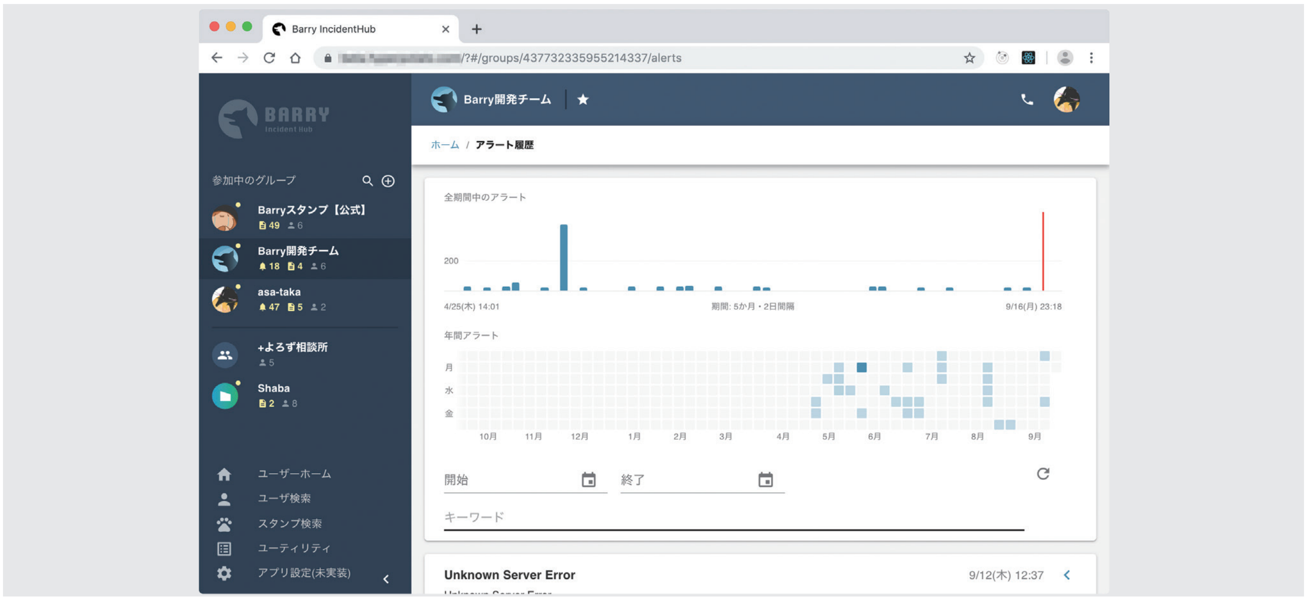


Figure 4: Graph of Alert Frequency

Stickers

We created emoji reactions and stickers for the incident comments to streamline communication (Figure 5). One problem is that expressions of gratitude and other emotional content in the form of comments increase the amount of information relating to an incident, making it hard to pick out the important details. With this in mind, we implemented an emoji reactions feature. We also created stickers like those used on social media, making it easy to convey basic/standard informational content.

Avatars

Avatars can be configured for each user and operations team. This feature is also widely used on social media services and helps improve visibility. The purpose is to prevent mistakes by allowing users and operations teams to freely configure their own avatars.

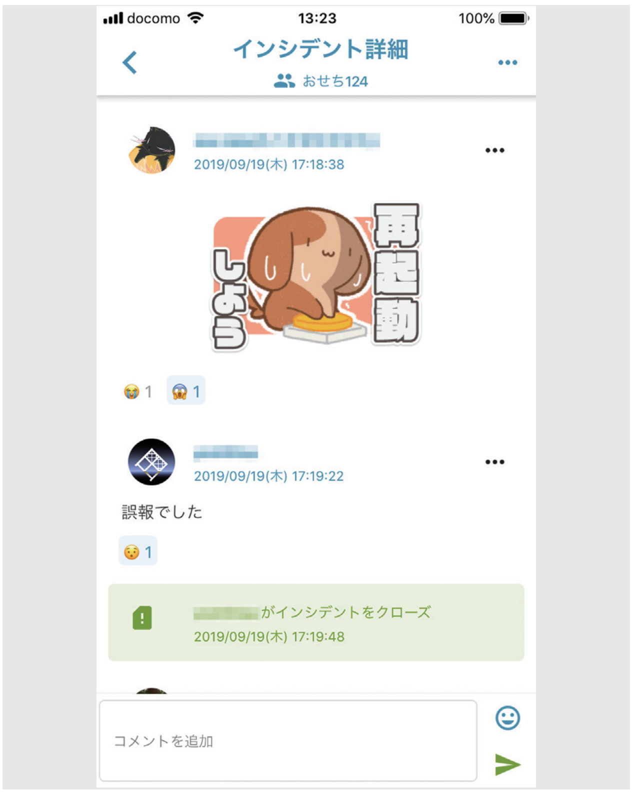
Webhook

The available features are also designed with automation in mind. An API is provided for everything that can be done via the screen, so users have the option of automating via software. Barry also has other automation features, notably webhook, which we implemented in response to user requests. Webhooks are a way for Web applications to provide information to external systems and are widely used by Web services and the like. Barry acts as the recipient of this information and thus supports the receipt of alerts and escalation initiations. Specifically, linking to webhooks such as Grafana makes it possible to link into existing systems without additional development. We also created a command line tool, so Barry can be used via simple scripts. We expect these features to be used in automating the work involved in dealing with faults.

3.4 Using Barry to Deal with Faults

Now let's follow the system operations process with Barry deployed.

Barry is an operations system for use within IIJ, and service operators perform the following initial setup.



[redacted]
2019/09/19(木) 17:18:38

[redacted]
2019/09/19(木) 17:19:22
誤報でした

[redacted]がインシデントをクローズ
2019/09/19(木) 17:19:48

Figure 5: Example of Emoji and Stickers on the Comment Screen

1. Set Barry to be the destination for service monitoring alerts
2. Define rules to say who is called and in what order when an issue is escalated
3. Troubleshooters install the Barry mobile app on their smartphones

With these arrangements in place, service monitoring alerts are sent to Barry when they occur. Upon receiving an alert, Barry saves the details, determines which operations team to use, and escalates the issue. The escalation process involves ringing people's smartphones according to the calling rules defined for the chosen operations team(s) until a troubleshooter is found.

Users learn of the escalation when their smartphones ring and then check why it was raised. The notification includes details of the alert, and if they are able to deal with the issue, users reply via the app to say they will start working on it. The system ends the calling process at this point, and the group is notified that responders have been selected.

The escalation feature is done with its role at this point, and the focus shifts to the incident features that provide integrated information tracking. The troubleshooters go over the event based on the information in the alert and put this information together into an incident. They then start working on rectifying the fault, leaving comments as they go. Information is shared within the operations team as it is added, including notifications to the mobile app, and people other than the designated troubleshooters can also add comments as necessary. Additional people can also be called on if the troubleshooters are unable to handle the issue alone.

Once the fault has been dealt with, the incident is updated as complete and Barry's work is done. The information recorded on the incident and the escalation history are stored in the system. A search feature is also available, so responders can refer to how similar issues were handled in the past as they work to fix a fault.

3.5 Operations

Barry's system is needed when dealing with faults in a range of services, so it needs to have high availability. Naturally, Barry itself can also fail, so it is designed and operated on the assumption that faults will occur.

The system is structured to have three independent regions, two of which provide redundancy for a single Barry system (Figure 6). The remaining region runs a separate Barry system. This is used by Barry's operators and comes into play when dealing with faults in Barry itself.

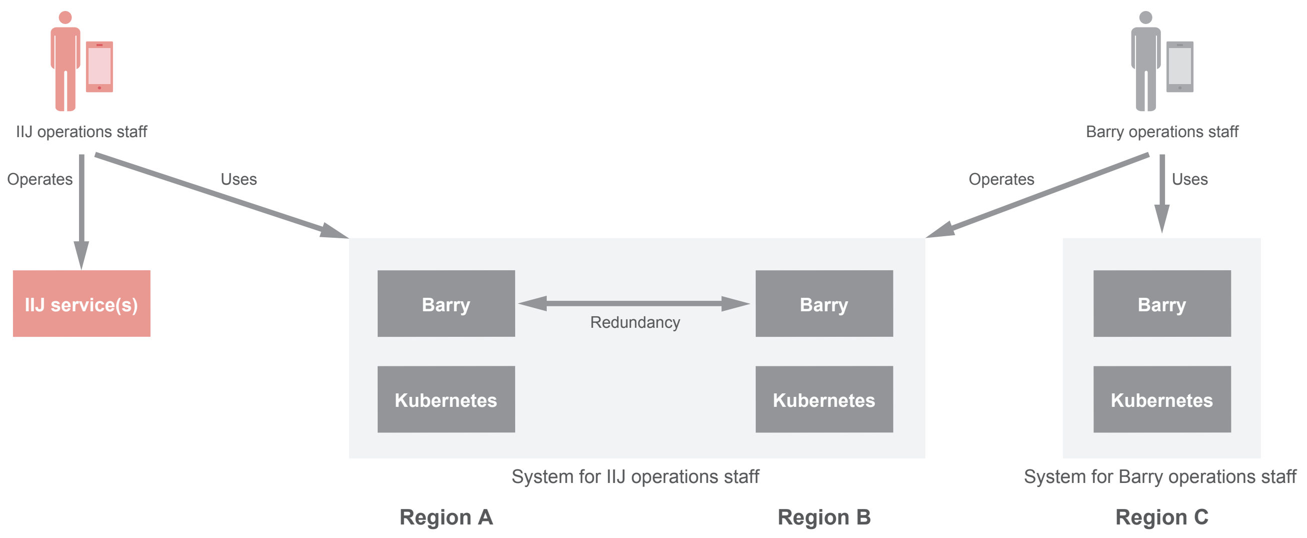


Figure 6: Barry System Structure

An independent Kubernetes cluster is run in each region, and Barry runs atop Kubernetes. Configuring the system to use Kubernetes' features eliminates the need to worry about hardware failures when running Barry.

Barry uses an external service to send notifications to smartphones. It is implemented as a combination of multiple external services so that a failure or delay in an external service does not become a single point of failure (Figure 7). The server looks at the responses of smartphones to which notifications are sent, and if an anomaly in the notification system is detected, Barry automatically falls back to using automated phone calls.

In the event of a top-level domain failure, the service may become inaccessible due to a name resolution failure, even if Barry is operating normally. To address this, we have set up multiple domains to ensure service access redundancy.

While a little different from system faults, we also deal with mobile app problems. On the server side, operators can roll back when problems occur, but they are not able to deal with issues in the apps installed on individual devices. Fatal errors cause the calling functionality to stop working, so two versions of the app are distributed. Along with a normal version of the app that is updated from time to time, an emergency version that is confirmed to be stable can also be installed.

3.6 Deployment and Impact

We released Barry internally in July 2020. Replacing the entire fault response system all at once would not be realistic, so our approach since the release has been to switch individual services over to Barry for the teams that want it. On the user end, there was the need to replace the mechanism by which alerts are sent and so forth, and this work of switching things over is progressing with help from the service teams. It's quite easy to adopt Barry particularly for newly launching services.

There are also tools created by Barry users now, so we have a real sense that the decision to open up the API is helping to facilitate the automation and streamlining of work for users.

Barry enables continuous calling like phone calls while also efficiently communicating information by sending a text message at the same time. Automation makes it possible to leave the simple procedure of calling people up to Barry. And the operating structure increases parallelism in the calling process, so candidates can be contacted all at once. When phoning people one after the other in sequence, we

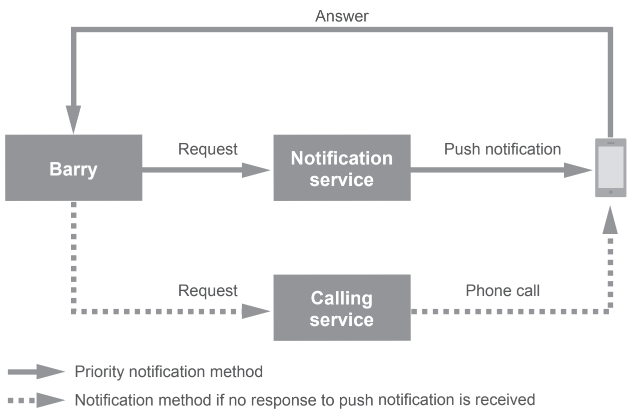

Answer
Barry
Request
Notification service
Push notification
Request
Calling service
Phone call
Priority notification method
Notification method if no response to push notification is received

Figure 7: Fallback for Barry System Faults

experienced delays in initiating the response when only some of the candidates were able to deal with the issue, and we have been able to resolve this as well. One of our tasks was to speed up the initial part of the troubleshooting process, so the system helps with this.

There are currently 633 users and 190 operations teams. We conducted a post-release user questionnaire about using Barry. We asked whether frequency of use and Barry's introduction had improved the way they work, and we also asked what users would like to see improved. I will go over these topics below.

As to what had improved, the responses mentioned the speeding up of the initial response and the ability to see current status, which were tasks we had identified. Having the system make the calls has reduced the workload, and automating the process from alert to escalation means that people can find out about faults happening earlier. The responses also mentioned communication. Because it is now easier to tell what the status of the fault response is within operations teams, people are finding it easier to coordinate their efforts. The positive feedback on improvements flowing from Barry's introduction indicates to us that it is lending a hand on the operations front.

Meanwhile, some people have also asked for improvements to Barry.

One request is to simplify links with existing systems. We have provided an API and designed Barry to be suitable for a range of use cases, but modifications do need to be made to existing systems in order to use Barry. Users have asked us for a way to get started using Barry with only minimal changes to existing operations systems. The system is designed to work within IIJ's own unique set of circumstances, so we plan to address such individual requests in a flexible manner going forward.

Another was to address concerns about stability. As discussed, Barry is a system that is used when faults occur, so it needs to be stable. One criterion users look at when assessing a system for adoption is its track record in operation, but having been released not long ago, Barry lacks an adequate track record. To ensure people can use the system with peace of mind, a priority for us in providing Barry is to build up this sort of stable track record ahead.

Deploying Barry internally was a major milestone for us, but we still have work to do. We hope to contribute to maintaining and enhancing the quality of IIJ services by continuing to update the system going forward.

Yushi Nakai
Operation System Development Section, Operation Engineering Department, Infrastructure Engineering Division, IIJ
Mr. Nakai joined IIJ in 2007. He is involved in the development of services and operations systems.