CHAPTER 13

Where Intelligence Lives

When building an Intelligent System you'll need to decide where the intelligence should live. That is, where you will bring the model, the runtime, and the context together to produce predictions—and then how you will get those predictions back to the intelligent experience.

The runtime could be located in the user's device, in which case you'll need to figure out how to update models across your user base.

The runtime could be in a service you run, in which case you'll have to figure out how to get the context (or features) from the user's device to the service cheaply enough, and with low enough latency, to make the end-to-end experience effective.

This chapter discusses how to decide where your intelligence should live. It starts by discussing the considerations for deciding where intelligence should live, including latency and cost, and why these matter for various types of intelligent experiences. It then discusses common patterns for positioning intelligence across clients and services, including the pros and cons of each option.

Considerations for Positioning Intelligence

Some key considerations when deciding where intelligence should live are these:

- Latency in updating the intelligence.
- Latency in executing the intelligence.
- The cost of operating the intelligence.
- What happens when users go offline.

In general, you will have to make trade-offs between these properties. This section discusses each of them in turn.

G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_13

Latency in Updating

One consideration when deciding where to position intelligence is the latency you will incur when you try to update the intelligence. New intelligence cannot benefit users until it gets to their runtime and replaces the old intelligence.

The latency in updating intelligence can be very short when the runtime is on the same computer as the intelligence creation environment; the latency can be very long when the runtime is on a client computer, and the client computer does not connect to the Internet very often (and so it can only get new intelligence once in a while).

Latency in transporting intelligence to a runtime is important when:

- The quality of intelligence is evolving quickly.
- The problem you are trying to solve is changing quickly.
- There is risk of costly mistakes.

We'll discuss these situations where latency in updating can cause problems in more detail, including examples of how they can impact intelligent experiences.

Quality Is Evolving Quickly

Latency in updating intelligence matters when the quality of the intelligence is evolving quickly.

When a problem is new, it is easy to make progress. There will be all sorts of techniques to explore. There will be new data arriving every day, new users using the product in ways you didn't anticipate. There will be excitement—energy. The quality of intelligence might improve rapidly.

For example, imagine it's day one—you've just shipped your smart shoe. This shoe is designed to adjust how tight it is based on what its wearer is doing. Sitting around doing nothing? The shoe automatically relaxes, loosening the laces so the wearer can be more comfortable. But if the user starts running? Jumping? The shoe automatically tightens the laces, so their wearer will get better support and be less likely to get an injury.

So you've had a huge launch. You have tens of thousands of smart shoes in the market. You are getting telemetry for how users are moving, when they are overriding the shoe to make their laces tighter, and when they are overriding the shoe to make their laces looser. You are going to want to take all this data and produce new intelligence.

If you produce the new intelligence, test it, and determine it isn't much better than the intelligence you shipped with—you're fine! No need to worry about latency in updating intelligence, because you don't have any benefit to gain.

But if the new intelligence is better—a lot better—you are going to be desperate to get it pushed out to all the shoes on the market. Because you have users wearing those shoes. They'll be talking to their friends. Reviewers will be writing articles. You are going to want to update that intelligence quickly!

When intelligence is improving quickly, latency in deployment will get in the way of taking advantage of the intelligence. This is particularly important in problems that are very big (with many, many contexts) or problems that are very hard to solve (where quality will be improving for a long time to come).

Problem Changing Quickly

Latency in updating intelligence matters when the problem is changing quickly, because it is open-ended or changing over time.

In some domains new contexts appear slowly, over the course of weeks or months. For example, new roads take time to build; tastes in music evolve slowly; new toaster products don't come onto the market every day. In these cases having days or weeks of latency in deploying new intelligence might not matter.

On the other hand, in some domains new contexts appear rapidly, hundreds or thousands of times per day. For example, new spam attacks happen every second; hundreds of new news articles are written per day; hurricanes make landfall; stock markets crash. In these cases, it might be important to be able to deploy new intelligence fast.

There are two important aspects to how problems change:

1. How quickly do new contexts appear?
2. How quickly do existing contexts disappear?

Erosion will happen slowly in domains where new contexts are added, but old contexts remain relevant for a reasonably long time. For example, when a new road is constructed your intelligence might not know what to do with it, but your intelligence will still work fine on all the existing roads it does know about.

Erosion will happen quickly in domains where new contexts displace old ones, and the intelligence you had yesterday is no longer useful today. For example, when a new song comes onto the top of the charts, an old song leaves. When a spammer starts their new attack, they stop their old one.

When a problem changes frequently in ways that erode the quality of existing intelligence, latency in updating intelligence can be a critical factor in the success of an Intelligent System.

Risk of Costly Mistakes

Latency in updating intelligence matters when the Intelligent System can make costly mistakes that need to be corrected quickly.

Intelligence makes all kinds of mistakes. Some of these mistakes aren’t too bad, particularly when the intelligent experience helps users deal with them. But some mistakes can be real problems.

Consider the smart-shoe example, where laces automatically loosen or tighten based on what the wearer is doing. Imagine that some small percent of users fidget in a particular way that makes the shoe clamp down on their feet, painfully.

Or imagine that a small percentage of users play right field on a baseball team. Ninety-nine percent of the time they are standing there, looking at the clouds go by—and then one percent of time they are sprinting wildly to try to catch a fly ball. Maybe the smart-shoes can’t keep up. Maybe right fielders are running out of their shoes, slipping, and getting hurt.

When your Intelligent System’s mistakes have high cost, and these costs can’t be mitigated with a good user experience, latency can be very painful. Imagine users calling, complaining, crying about the damage you’ve caused them, day after day, while you wait for new intelligence to propagate. It can make you feel bad. It can also put your business at risk.

Latency in Execution

Another consideration for deciding where intelligence should live is the latency in executing the intelligence at runtime.

To execute intelligence, the system must gather the context, convert the context to features, transport the features to where the intelligence is located (or sometimes the whole context gets transported and feature extraction happens later), wait for the intelligence to execute, wait for the result of the intelligence execution to get back to the experience, and then update the experience. Each of these steps can introduce latency in executing the intelligence.

The latency in execution can be short when the intelligent runtime lives on the same computer as the intelligent experience; it can be long when the intelligent runtime and intelligent experience live on different computers.

Latency in intelligence execution can be a problem when:

- Users will have to wait for the latency.
- The right answer changes quickly and drastically.

Latency and its effects on users can be difficult to predict. Sometimes users don't care about a little bit of latency. Sometimes a little latency drives them crazy and totally ruins an experience. Try to design experiences where latency trade-offs can be changed easily (during orchestration) so various options can be tested with real users.

Latency in Intelligent Experience

Latency in execution matters when users notice it, particularly when they have to wait for the latency before they can continue. This can occur when:

- **The intelligence call is an important part of rendering the experience**. Imagine an application that requires multiple intelligence calls before it can properly render. Maybe the application needs to figure out if the content is potentially harmful or offensive for the user before rendering it. If there is nothing for the user to do while waiting for the intelligence call, then latency in intelligence execution could be a problem.
- **The intelligence call is interactive**. Imagine an application where the user is interacting directly with the intelligence. Maybe they throw a switch, and they expect the light to turn on instantly. If the switch needs to check with the intelligence before changing the light, and the intelligence call takes hundreds or thousands of milliseconds— users might stub their toes.

On the other hand, intelligent experiences that are intrinsically asynchronous, such as deciding whether to display a prompt to a user, are less sensitive to latency in execution.

The Right Answer Changes Drastically

Latency in executing intelligence matters when the right answer changes rapidly and drastically.

Imagine creating an intelligence to fly a drone. The drone is heading to an objective; the correct answer is to fly straight at the objective. No problem. And then someone steps in front of the drone. Or imagine the drone flying on a beautiful, blue, sunny day and then a huge gust of wind comes out of nowhere. In these situations, the right control for the drone changes, and changes quickly.

When the right course of action for an Intelligent System changes rapidly and drastically, latency in executing intelligence can lead to serious failures.

Cost of Operation

Another consideration when deciding where intelligence should live is the cost of operating the Intelligent System.

Distributing and executing intelligence takes CPU, RAM, and network bandwidth. These cost money. Some key factors that can drive the cost of an Intelligent System include:

- The cost of distributing intelligence.
- The cost of executing intelligence.

The Cost of Distributing Intelligence

Distributing intelligence costs both the service and the user money, usually in the form of bandwidth charges. Each new piece of intelligence needs to be hosted (for example on a web service), and the runtime must periodically check for new intelligence, and then download any it finds. This cost is proportional to the number of runtimes the intelligence must go to (the number of clients or services hosting it), the size of the intelligence updates, and the frequency of the updates. For Internet-scale Intelligent Systems, the cost of distributing intelligence can be very large.

It's also important to consider costs for users. If the primary use case is for users on broadband, distributing models might not be a concern—it might not cost them much. But when users are on mobile devices, or in places where network usage is more carefully metered and billed, the bandwidth for intelligence distribution may become an important consideration.

The Cost of Executing Intelligence

Executing intelligence can also have a bandwidth cost when the intelligence is located in a service and clients must send the context (or features) to the service and get the response. Depending on the size of the context and the frequency of calls, it may cost more to send context to the service than to send intelligence to the client. Keep in mind that telemetry and monitoring will also need to collect some context and feature information from clients; there is opportunity to combine the work and reduce cost.

Executing intelligence also takes CPU cycles and RAM. Putting the intelligence runtime in the client has the advantage that users pay these costs. But some types of intelligence can be very expensive to execute, and some clients (like mobile ones) have resource constraints that make heavyweight intelligence runtimes impractical. In these cases, using intelligence runtimes in the service (maybe with customized hardware, like GPUs or FPGAs) can enable much more effective intelligences.

When considering the cost of operation, strive for an implementation that:

1. Is sensitive to the user and does not ask them to pay costs that matter to them (including bandwidth, but also power in a mobile device, and so on).
2. Does let users pay the parts of the cost they won't notice (so you don't have to buy lots of servers).
3. Balances all the costs of running the Intelligent System and scales well as the number of users and quality of intelligence grow.

Offline Operation

Another consideration when deciding where intelligence should live is whether the Intelligent System needs to function when it is offline (and unable to contact any services).

It isn't always important for an Intelligent System to work when it is offline. For example, a traffic prediction system doesn't need to work when its user is in an airplane over the Pacific Ocean. But sometimes Intelligent Systems do need to work offline—for example, when the intelligence runs in a restricted environment (like a military vehicle) or in a life-critical system.

When it is important to function offline, some version of the intelligence must live in the client. This can be the full intelligence, or it can be more like a backup—a reduced version of the overall intelligence to keep users going while the service comes back online.

Places to Put Intelligence

This section explores some of the options for positing intelligence in more detail. It introduces some common patterns, including:

- Static intelligence in the product
- Client-side intelligence
- Server-centric intelligence
- Back-end (cached) intelligence
- Hybrid intelligence

This section discusses each of these approaches and explores how well they address the four considerations for where intelligence should live: latency in updating, latency in execution, cost of operation, and offline operation.

Static Intelligence in the Product

It's possible to deliver intelligence without any of this Intelligent System stuff. Simply gather a bunch of training data, produce a model, bundle it with your software, and ship it.

This is very similar to shipping a traditional program. You build it, test it as best you can—in the lab, or with customers in focus groups and beta tests—tune it until you like it, and send it out into the world.

The advantages of this is that it is cheaper to engineer the system. It might be good enough for many problems. It can work in situations without the ability to close the loop between users and intelligence creation. And there is still the possibility for feedback (via reviews and customer support calls), and to update the intelligence through traditional software updates.

The disadvantage is that intelligence updates will be more difficult, making this approach poorly suited to open-ended, time-changing, or hard problems.

Latency in Updating Intelligence: Poor

Latency in Execution: Excellent

Cost of Operation: Cheap

Offline Operation: Yes

Disadvantage Summary: Difficult to update intelligence. Risk of unmeasurable intelligence errors. No data to improve intelligence.

Client-Side Intelligence

Client-side intelligence executes completely on the client. That is, the intelligence runtime lives fully on the client, which periodically downloads new intelligence.

The download usually includes new models, new thresholds for how to interpret the models' outputs (if the intelligence is encapsulated in an API—and it should be), and (sometimes) new feature extraction code.

Client-side intelligence usually allows relatively more resources to be applied to executing intelligence. One reason for this is that the intelligence can consume idle resources on the client at relatively little cost (except maybe for power on a mobile device). Another reason is that the latency of the runtime is not added to any service call latency, so there is relatively more time to process before impacting the experience in ways the user can perceive.

The main challenge for client-side intelligence is deciding when and how to push new intelligence to clients. For example, if the intelligence is ten megabytes, and there are a hundred thousand clients, that's about a terabyte of bandwidth per intelligence update. Further, models don't tend to compress well, or work well with incremental updates, so this cost usually needs to be paid in full.

Another potential complexity of client-side intelligence is dealing with different versions of the intelligence. Some users will be offline, and won't get every intelligence update you'd like to send them. Some users might opt-out of updates (maybe using firewalls) because they don't like things downloading to their machines. These situations will make interpreting user problems more difficult.

Another disadvantage of client-side models is that they put your intelligence in the hands of whoever wants to take a look at it: maybe a competitor, maybe someone who wants to abuse your service, or your users—like a spammer. Once someone has your model they can run tests against it. They can automate those tests. They can figure out what type of inputs gets the model to say one answer, and what type of inputs gets it to say another. They can find the modifications to their context (e-mail, web page, product, and so on) that trick your model into making exactly the type of mistake they want it to make.

Latency in Updating Intelligence: Variable

Latency in Execution: Excellent

Cost of Operation: Based on update rate.

Offline Operation: Yes

Disadvantage Summary: Pushing complex intelligence to clients can be costly. Hard to keep every client in-sync with updates. Client resources may be constrained. Exposes the intelligence to the world.

Server-Centric Intelligence

Server-centric intelligence runs in real-time in the service. That is, the client gets the context (or features) and sends them to the server, and the server executes the intelligence on the features and returns the result to the client.

Using server-centric intelligence allows models to be updated quickly, and in a controlled fashion, by pushing new models to the server (or servers) running the intelligence. It also makes telemetry and monitoring easier because much of the data to log will already be in the service as part of the intelligence calls.

But server-centric intelligence needs to be scaled as the user base scales. For example, if there are a hundred intelligence request per second, the service must be able to execute the intelligence very quickly, and probably in parallel.

Latency in Updating Intelligence: Good

Latency in Execution: Variable, but includes Internet round-trip.

Cost of Operation: Service infrastructure and bandwidth can have significant cost; may cost users in bandwidth.

Offline Operation: No

> **Disadvantage Summary:** Latency in intelligence calls. Service infrastructure and bandwidth costs. User bandwidth costs. Cost of running servers that can execute intelligence in real time.

Back-End (Cached) Intelligence

Back-end intelligence involves running the intelligence off-line, caching the results, and delivering these cached results where they are needed. Cached intelligence can be effective when analyzing a finite number of things, like all the e-books in a library, all the songs a service can recommend, or all the zip codes where an intelligent sprinkler is sold. But back-end intelligence can also be used when there aren't a finite number of things, but contexts change slowly.

For example, a sprinkler is sold into a new zip code. The service has never considered that zip code before, so it returns some default guess at the optimal watering time. But then the back-end kicks off, examines all the info it can find about the zip code to produce a good watering plan and adds this watering plan to its cache. The next time a sprinkler is sold in that zip code, the service knows exactly how to water there (and maybe the system even updates the watering plan for that poor first guy who kicked off the whole process).

Back-end Intelligent Systems can afford to spend more resources and time on each intelligence decision than the other options. For example, imagine a super complex watering-plan model that runs for an hour on a high-end server to decide how to water in each zip code. It analyzes satellite images, traffic patterns, the migration of birds and frogs in the regions—whatever it takes. Such a model might take months to run on an embedded computer in a sprinkler—impractical. It can't run on a server that needs to respond to hundreds of calls per second—no way. But it can run in back-end 'every so often' and the results of its analysis can be cached.

Intelligence caches can live in services; parts of them can be distributed to clients too.

One disadvantage of back-end intelligence is that it can be more expensive to change models, because all of the previous cached results might need to be recomputed.

Another disadvantage is that it only works when the context of the intelligence call can be used to "look up" the relevant intelligence. This works when the context describes an entity, such as a web page, a place, or a movie. It doesn't work when the context describes less-concrete things, like a user-generated block of text, a series of outputs from a sensor-array, or a video clip.

Latency in Updating Intelligence: Variable

Latency in Execution: Variable

Cost of Operation: Based on usage volume

Offline Operation: Partial

Disadvantage Summary: Not effective when contexts change quickly, or when the right answer for a context changes quickly. Can be expensive to change models and rebuild the intelligence caches. Restricts intelligence to things that can be looked up.

Hybrid Intelligence

In practice it can be useful to set up hybrid intelligences that combine several of these approaches.

For example, a system might use a back-end intelligence to deeply analyze popular items, and a client-side intelligence to evaluate everything else.

Or a system might use a client-side intelligence in most cases, but double-check with the service when a decision has serious consequences.

Hybrid intelligences can mask the weaknesses of their various components.

But hybrid intelligences can be more complex to build and to orchestrate. Consider, if the system gives an incorrect answer, what part of the intelligence did the mistake come from? The client-side model? The intelligence that was cached in the service? Some subtle interaction between the two?

Sometimes it's even hard to know for sure what state all of those components were in at the time of the mistake.

Nevertheless, most large Intelligent Systems use some form of hybrid approach when determining where their intelligence should live.

Summary

Choosing where your intelligence will live is an important part of creating a successful Intelligent System. The location of intelligence can affect:

- **The latency in updating the intelligence:** This is a function of how far the intelligence needs to move to get from the creation environment to the runtime and how often the runtime is online to take an update.
- **The latency in executing the intelligence:** This is a function of moving the context and the features from the intelligent experience to the intelligence runtime and moving the answer back.
- **The cost of operating the Intelligent System:** This is a function of how much bandwidth you need to pay for to move intelligence, context, and features and how much CPU you need to pay for to execute intelligence.
- **The ability of the system to work offline:** This is a function of how much of the intelligence can function on the client when it can't communicate with your service components.

There are many options for balancing these properties. Here are some common patterns:

- **Static intelligence:** This puts the intelligence fully in the client without connecting it to a service at all.
- **Client-side intelligence:** This puts the intelligence fully in the client, but connects it to a service for intelligence updates and telemetry.
- **Server-centric intelligence:** This puts the intelligence fully in a service and requires a service call every time intelligence needs to be executed on a context.

- **Back-end (cached) intelligence:** This executes intelligence offline on common contexts and delivers answers via caching.
- **Hybrid intelligence:** This is the reality for most large-scale Intelligent Systems and combines multiple of the other approaches to achieve the system's objectives.

For Thought…

After reading this chapter, you should:

- Know all the places intelligence can live, from client to the service back-end, and the pros and cons of each.
- Understand the implications of intelligence placement and be able to design an implementation that is best for your system.

You should be able to answer questions like these:

- Imagine a system with a 1MB intelligence model, and 10KB of context for each intelligence call. If the model needs to be updated daily, at what number of users/intelligence call volume does it make sense to put the intelligence in a service instead of in the client?
- If your application needs to work on an airplane over the Pacific Ocean (with no Internet), what are the options for intelligence placement?
- What if your app needs to function on an airplane, but the primary use case is at a user's home? What are some options to enable the system to shine in both settings?