Engineering

Introducing Domain-Oriented Microservice Architecture

July 23, 2020 / Global

Introduction

Recently there has been substantial discussion around the downsides of service oriented architectures and microservice architectures in particular. While only a few years ago, many people readily adopted microservice architectures due to the numerous benefits they provide such as flexibility in the form of independent deployments, clear ownership, improvements in system stability, and better separation of concerns, in recent years people have begun to decry microservices for their tendency to greatly increase complexity, sometimes making even trivial features difficult to build.

As Uber has grown to around 2,200 critical microservices, we experienced these tradeoffs first hand. Over the last two years, Uber has attempted to reduce microservice complexity while still maintaining the benefits of a microservice architecture. With this blog post we hope to introduce our generalized approach to microservice architectures, which we refer to as "Domain-Oriented Microservice Architecture" (DOMA).

While it's been popular in recent years to criticize microservice architectures because of these downsides, few people have advocated an outright rejection of microservice architectures. The operational benefits are too important, and it seems that there are no, or limited, alternatives. **Our goal with DOMA is to provide a way forward for organizations that want to reduce overall system complexity while maintaining the flexibility associated with microservice architectures.**

This piece explains DOMA, the concerns that led to the adoption of this architecture for Uber, its benefits for platform and product teams, and, finally, some advice for teams who want to adopt this architecture.

What is a microservice?

Microservices are an extension of service oriented architectures. As opposed to the fairly large "services" of the 2000s, microservices are applications that represent a set of narrowly scoped functionality. These applications are hosted and available over the network and expose a well-defined interface. Other applications call this interface by making a "remote procedure call" (RPC).

The key characteristic of microservice architecture is the way in which code is hosted, called, and deployed. If we think about large, monolithic applications, they are generally split into

encapsulated components with well-defined interfaces. These interfaces would then be called directly in-process as opposed to over the network. In this way, we can start to think of a microservice as a library with a performance hit (network I/O and serialization / deserialization) in order to call any of its functions.

When we think about microservices this way, we might question why we would adopt a microservice architecture at all. The answer is often *independent deployments and scaling*. With a large, monolithic application, an organization is forced to deploy or release all of their code at once. Each new version of an application can involve numerous changes. Deployments become risky and time consuming. Anyone can bring the whole system down.

In other words, organizations adopt microservices for an *operational* benefit at the expense of *performance*. Organizations also must take on the cost to maintain the infrastructure necessary to support microservices. In many situations, it turns out, this trade-off makes a lot of sense, but it is also a strong argument against a premature adoption of a microservice architecture.

Motivations

At Uber, we adopted a microservice architecture because we had (circa 2012-2013) primarily two monolithic services and ran into many of the operational issues that microservices solve.

- **Availability Risks.** A single regression within a monolithic code base can bring the whole system (in this case, all of Uber) down.
- **Risky, expensive deployments.** These were painful and time consuming to perform with the frequent need for rollbacks.
- **Poor separation of concerns.** It was difficult to maintain good separations of concerns with a huge code base. In an exponential growth environment, expediency sometimes led to poor boundaries between logic and components.
- **Inefficient execution.** These issues combined made it difficult for teams to execute autonomously or independently.

In other words, as Uber grew from 10s to 100s of engineers with multiple teams owning pieces of the tech stack, the monolithic architecture tied the fate of teams together and made it difficult to operate independently.

As a result, we adopted a microservice architecture. Ultimately our systems became more *flexible*, which allowed teams to be more *autonomous.*

- **System reliability.** Overall system reliability goes up in a microservice architecture. A single

service can go down (and be rolled back) without taking down the whole system.

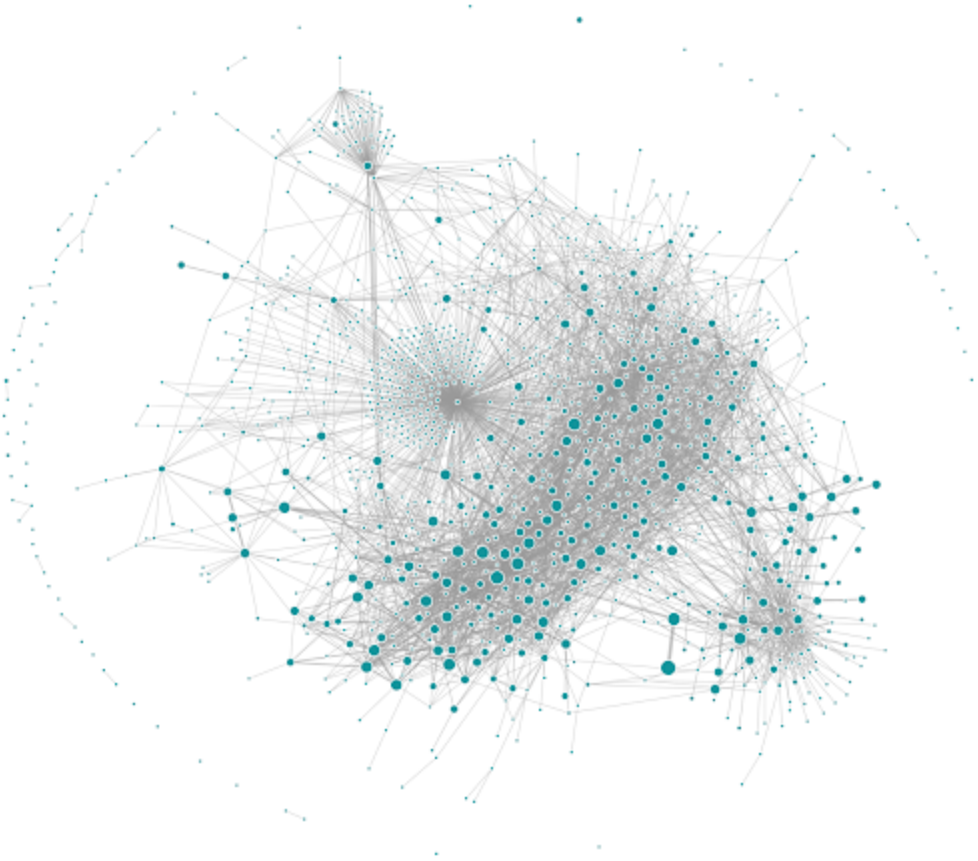
- **Separation of concerns.** Architecturally, microservice architectures force you to ask the question “why does this service exist?” more clearly defining the roles of different components.
- **Clear Ownership.** It becomes much clearer who owned what code. Services are typically owned at the individual, team, or org level enabling faster growth.
- Autonomous execution. Independent deployments + clearer lines of ownership unlock autonomous execution by various product and platform teams.
- **Developer Velocity.** Teams can deploy their code independently, which enables them to execute at their own pace.

It’s not an exaggeration to say that Uber would not have been able to accomplish the scale and quality of execution that we maintain today without a microservice architecture.

However, as the company grew even larger, 100s of engineers to 1000s, we began to notice a set of issues associated with greatly increased system *complexity*. With a microservice architecture one trades a single monolithic code base for black boxes whose functionality can change at any time and easily cause unexpected behavior.

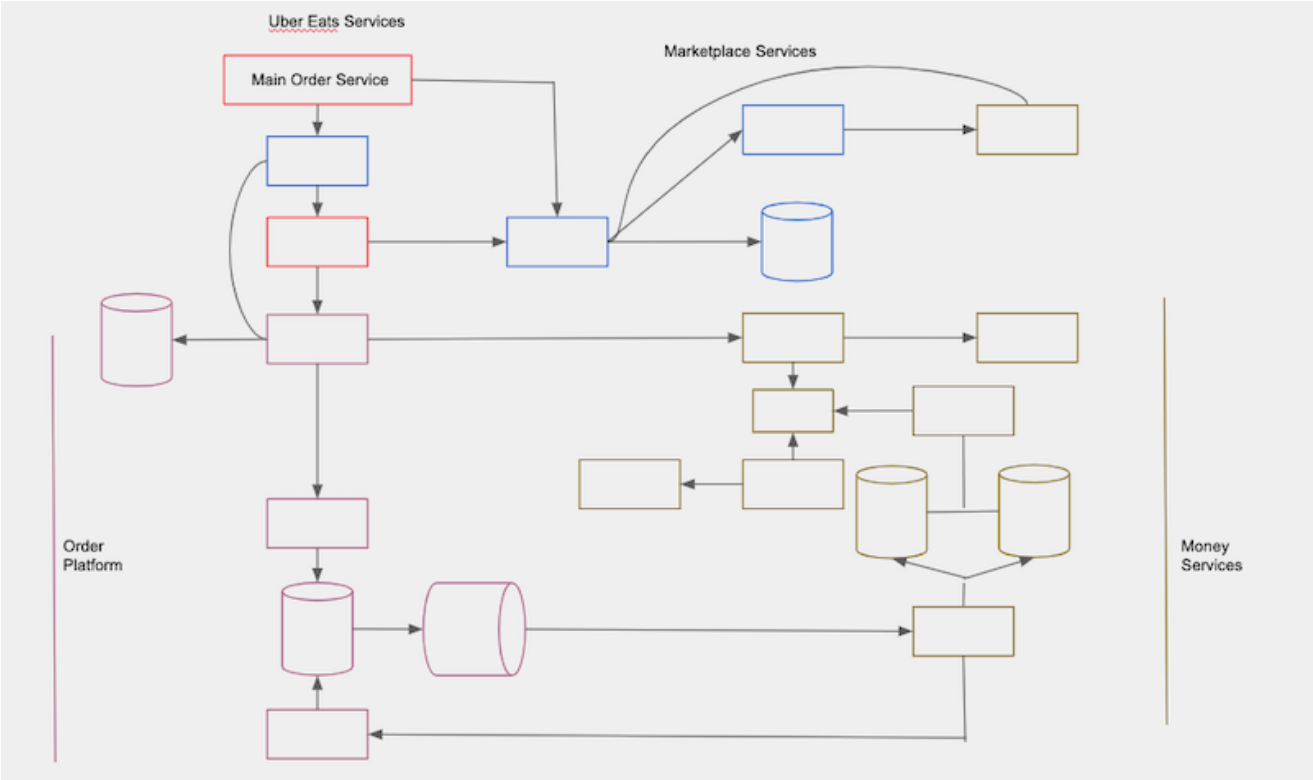
For instance, engineers had to work through around 50 services across 12 different teams in order to investigate the root cause of the problem.

Understanding dependencies between services can become quite difficult, as calls between services can go many layers deep. A latency spike in the nth dependency can cause a cascade of issues upstream. Visibility into what’s actually happening is impossible without the right tools, making debugging difficult.



Uber's microservice architecture circa mid-2018 from Jaeger

In order to build a simple feature an engineer often has to work across multiple services, all of which are owned by different individuals and teams. This requires extensive collaboration with time spent on meetings, design, and code review. The earlier promise of clear lines of service ownership is compromised as teams build code within each other's services, modify each other's data models, and even perform deployments on behalf of service owners. Networked monoliths can form, where services that appear to be independent all have to be deployed together to safely perform any change.



An example of a complex flow at Uber circa 2018, which required 10 touch points for a simple integration before DOMA.

The result is a slower developer experience, instability for service owners, more painful migrations, etc. For organizations that have already adopted a microservice architecture there is no turning back. It becomes a case of "*can't live with them, can't live without them*."

Domain-Oriented Microservice Architecture

If we can think of microservices as I/O bound libraries and a "microservice architecture" as a large, distributed application then we can use well understood architectures to think about how to organize our code.

"Domain-Oriented Microservice Architecture" thus draws heavily from established ways to organize code such as Domain-driven Design, Clean Architecture, Service-Oriented Architecture, and object- and interface-oriented design patterns. **We think of DOMA as innovative only insofar as it is a relatively novel way to leverage established design principles in large distributed systems in large organizations.**

The core principles and terminology associated with DOMA are as follows:

1. Instead of orienting around single microservices, we oriented around collections of related

microservices. We call these **domains.**

2. We further create collections of domains which we call layers. The layer that the domain belongs to establishes what dependencies the microservices within that domain are allowed to take on. We call this **layer design.**
3. We provide clean interfaces for domains that we treat as a single point of entry into the collection. We call these **gateways.**
4. Finally, we establish that *each domain should be agnostic to other domains,* which is to say, a domain shouldn't have logic related to another domain hard coded inside of its code base or data models. Since frequently teams do need to include logic in another team's domain (for example, custom validation logic or some meta context on a data model), we provide an **extension architecture** to support well defined extension points within the domain.

In other words, by providing a systematic architecture, domain gateways, and predefined extension points, DOMA intends to transform microservice architectures from something complex to something comprehensible: a structured set of flexible, reusable, and layered components.

The rest of this post digs into Uber's implementation of DOMA, the benefits we've seen, and practical advice for companies which might want to adopt this approach.

Uber's Implementation

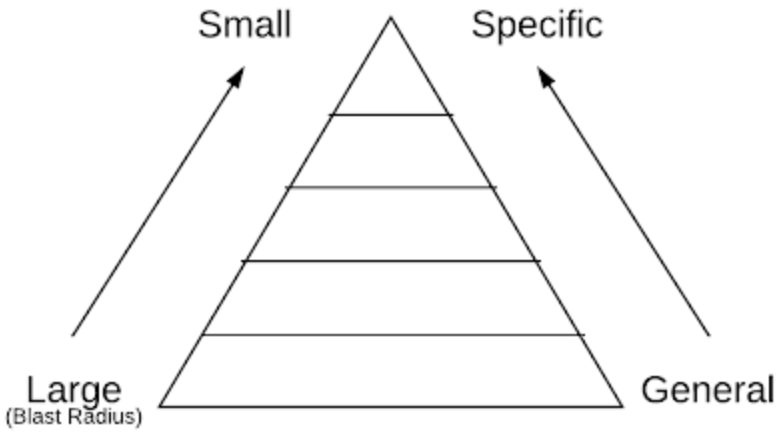
Domains

Uber domains represent a collection of one or more microservices tied to a logical grouping of functionality. A common question in designing a domain is "how big should a domain be?" We give no guidance here. Some domains can include tens of services, some domains only a single service. The important task is to think carefully about the *logical* role of each collection. For instance, our map search services constitute a domain, fare services are a domain, matching platform (matching riders and drivers) are a domain. These also don't always follow company org structure. The Uber Maps org itself is split into three domains, with 80 microservices behind 3 different gateways.

Layer Design

Layer design answers the question of "what service can call what other service?" within Uber's microservice architecture. As a result, we can think of layer design as "separation of concerns at scale." Alternatively, we can think of layer design as "dependency management at scale."

Layer design describes a mechanism for thinking about failure blast radius and product specificity across service dependencies at Uber. *As domains move from the bottom layer to the top layer, they impact fewer services in the case of an outage and represent more specific product use cases.* Conversely, functionality at the bottom layers have more dependents and as a result tend to have a larger blast radius and represent a more general set of business functionality. The figure below illustrates this concept.



One can think of the top layers as specific user experiences (such as mobile features), and the bottom layers as generalized business functionality (such as account management or marketplace trips). Layers only depend on the layers under them, which gives us a useful heuristic to think about questions like blast radius and domain integration.

It's worth noting that functionality often moves "down" this chart from specific to more general. One can imagine a simple feature that eventually becomes more and more of a platform as requirements evolve. In fact, this sort of migration downward is expected, and many of Uber's core business platforms started as rider or driver specific functionality that became more generalized as we developed more lines of business and they took on more dependencies (such as Uber Eats or Uber Freight).

Within Uber, we established the following five layers.

1. **Infrastructure layer.** Provides functionality that any engineering organization could use. It's Uber's answer to the big engineering questions, such as storage or networking.
2. **Business layer.** Provides functionality that Uber as an organization could use, but that is not specific to a particular product category or line of business (LOB) such as Rides, Eats, or Freight.
3. **Product layer.** Provides functionality that relates to a particular product category or LOB, but is agnostic to the mobile application, such as the "request a ride" logic which is

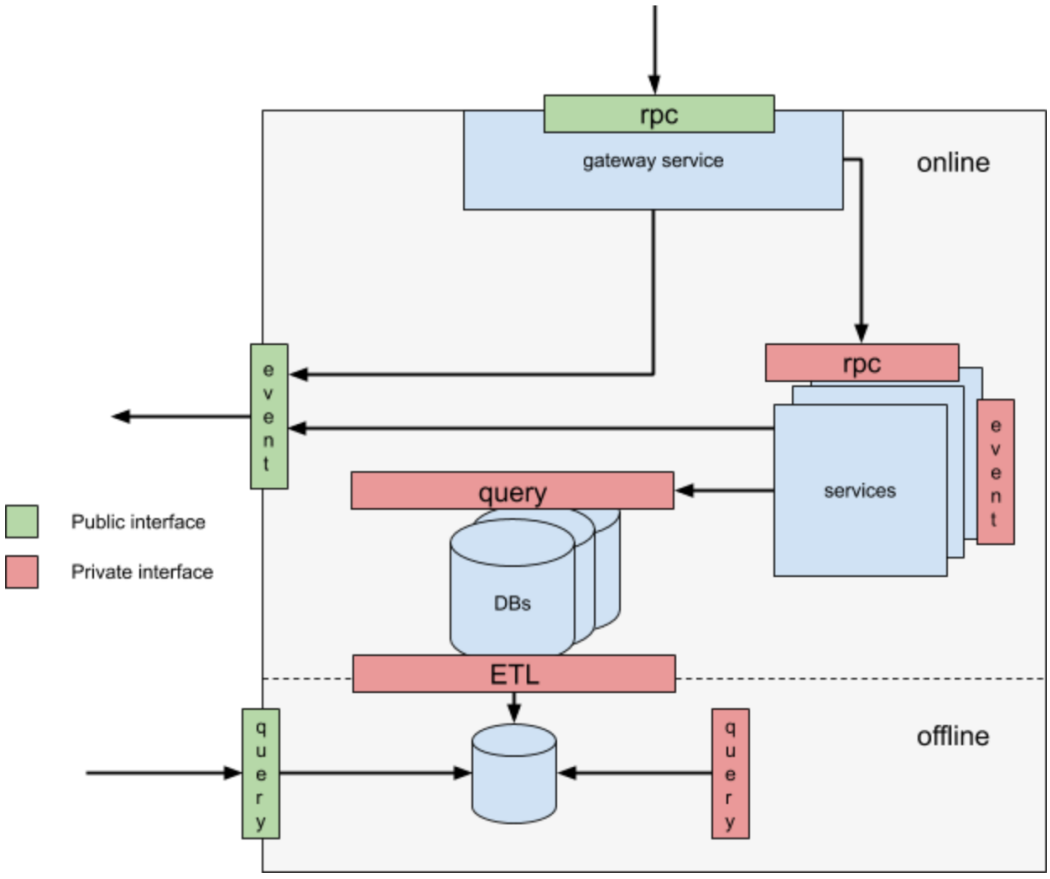
leveraged by multiple Rides facing applications (Rider, Rider “Lite”, m.uber.com, etc).

4. **Presentation.** Provide functionality that directly relates to features that exist within a consumer-facing application (mobile/web).
5. **Edge Layer.** Safely exposes Uber services to the outside world. This layer is also mobile application aware.

As you can see, each subsequent layer represents an increasingly specific grouping of functionality, and has a smaller and smaller blast radius (or, in other words, less components depend on the functionality within that layer).

Gateways

The term “Gateway API” is already a broadly established concept within microservice architectures. Our definition does not vary greatly from the established definition, except that we tend to think of gateways exclusively as a **single entry-point** into a collection of underlying services, which we call a **domain.** The success of a gateway relies on the success of the API design.



The following figure illustrates the high level diagram of a gateway. It abstracts away the internal details of the domains – multiple services, data tables, ETL pipelines etc. Only the interfaces –

RPC APIs, messaging events and queries are exposed to other domains.

Since upstream consumers only operate on a single service, gateways provide numerous benefits in terms of **future migrations, discoverability,** and overall reduction in **system complexity** with upstream services only taking a single dependency as opposed to dependencies on several downstream services that might exist within a domain. If we think about gateways in the sense of OO design, they are interface definitions, which enable us to do whatever we want in terms of the underlying "implementation" (in this case the collection of underlying microservices).

Extensions

Extensions represent a mechanism to *extend* domains. The basic definition of an extension is that it provides a mechanism for extending the functionality of an underlying service without changing the actual implementation of that service and without impacting its overall reliability. At Uber we provide two different extension models: **logic extensions** and **data extensions**. The concept of extensions has allowed us to scale our architecture to multiple teams being able to work independently of each other.

Logic Extensions

Logic extensions provide a mechanism for extending the underlying logic of a service. For logic extensions we use a variation of a *provider* or *plugin* pattern with an interface defined on a service-by-service basis. This makes it so that extending teams can implement extension logic in an interface-driven way without modifying the core code of the underlying platform.

For example, a driver goes online. Typically, we make various checks to ensure that a driver is allowed to go online (safety checks, compliance, etc.). Each of these is owned by an individual team. One way to implement this would be to have each team write logic in the same endpoint, but this can introduce complexity. Each check would require custom, and entirely unrelated, logic.

In the case of logic extensions, the "go online" endpoint would define an interface that they expect each extension to conform to with a predefined request type and a response. Each team would register an extension that would be responsible for the execution of this logic. In this case, they might simply take some context about the driver and return a bool, saying if the driver can go online or not. The go online endpoint would simply iterate through these responses, and determine if any of them are false.

This decouples the core code from each extension, and provides isolation between extensions, which don't know what other logic is executing. It's easy to build up more functionality around this, such as observability or feature flagging.

Data Extensions

Data extensions provide a mechanism for attaching arbitrary data to an interface to avoid bloat in core platform data models. For data extensions, we leverage Protobuf's Any functionality so that teams can add arbitrary data to requests. Services will often store this data or pass it to a logic extension so that the core platform is never responsible for deserializing (and thus "knowing about") this arbitrary context. Protobuf's Any implementation comes with some infrastructure overhead in exchange for stronger typing. For a simpler implementation, one could just as easily use a JSON string to represent arbitrary data.

```
// Totally made up example of a document type before data extensions

message Document {
  string uuid = 1;
  string type = 2;
  oneof extended  {
   driver.DriversLicense drivers_license = 3;
   eats.HealthInspection health_inspection = 4;
   // ... sometimes in the 10s of types here
  }
}
```

```
// Totally made up example of a simplified interface

message Document {
 string uuid = 1;
 string type = 2;
 google.protobuf.Any extension = 100;
}
```

Custom

Outside of logic and data extensions, many teams at Uber have introduced their own extension patterns that are appropriate for their domain. For example, much of the integrations tied to our presentation architecture uses DAG based task execution logic.

Benefits

Almost every major domain at Uber has been influenced on some level by DOMA. Over the last

year, we have focused primarily on Uber's business layer which provides generalized logic for each of our various lines of business.

DOMA is still young at Uber, and we are excited to share more data and in-depth examples of our architecture in the future. However, early signs have been extremely positive in terms of a simplified developer experience and a reduction in overall system complexity.

Products & Platforms

DOMA was the result of a consensus effort across product and platform teams at Uber. Platform support costs often dropped an order of magnitude. Product teams benefited from guard rails and accelerated development.

For example, an early platform consumer of our extensions architecture was able to drop the time to prioritize and integrate a new feature from three days to three hours by adopting an extension architecture with reduced time for code review, planning, and learning curve for consumers.

Reduced Complexity

Previously product teams would have to call numerous downstream services to leverage a domain; they now have to call just one. By reducing the number of touchpoints to onboard a new feature, platforms were able to reduce onboarding time by 25-50%. Furthermore, we were able to classify 2200 microservices into 70 domains. Roughly 50% of which have been implemented, and most of which have some plan for future adoption.

Future Migrations

At Uber, we calculated that the half-life of a microservice was 1.5 years, which means that every 1.5 years 50% of our microservices churn. Without gateways it's easy for a microservice architecture to fall into a "migration hell" as a result of this churn. Ever changing microservices constantly require upstream migrations. Gateways enable teams to avoid dependencies on the underlying domain services, which means those services can change without forcing an upstream migration.

Two of Uber's largest platform rewrites in the last year happened behind gateways. These platforms had hundreds of services that depended on them that would have had to migrate existing consumers. The cost of migration in these cases would have been extremely high, making a complete platform rewrite infeasible.

New Lines of Business & Products

Platforms designed using DOMA have proven to be much more extensible and easier to maintain. Most teams at Uber who adopted DOMA did so because supporting new lines of business had become too expensive.

Practical Advice

This section provides some practical advice for companies that might want to adopt DOMA. The guiding principle here is that in our experience a mature and thoughtful microservice architecture stems from quiet nudges in the right direction at the right time. The reality is that a true “rewrite” is never possible for one’s entire microservice architecture.

As a result, we think of evolving a microservice architecture more like “trimming a hedge” so that it eventually grows correctly, rather than a top-down or one-time architecture (or re-architecture) effort. It’s a dynamic and progressive process.

Startups

The driving questions should be “when should we adopt a microservice architecture?” and “does it make sense for our organization?” As we’ve seen above, while microservices provide an operational benefit to organizations with a large number of engineers, this trades off with an increase in complexity that can make features more difficult to build.

In small organizations, the operational benefit likely does not offset the increase in architectural complexity. Furthermore, microservice architectures often require dedicated engineering resources to support which may be out of budget for an early stage company or else suboptimal from a prioritization perspective.

With this in mind, it isn’t unreasonable to hold off on microservices altogether for some time. If an organization does choose to adopt microservices, it should think about the “microservice as large distributed application” analogy, and the separation of concerns between microservices it wants to build. Also, recognize that the first microservices will likely be the most important and longest lasting as they truly describe the core of the business.

Midsize

Once a company becomes midsized with multiple teams and the clear separation of concerns becomes hazy between different features and platforms, microservice architectures become

more obviously useful.

It's at this stage that one can begin to think about hierarchies between microservices. Dependency management may become more important, as some services begin to become more obviously critical to business operation, and more and more teams rely on them.

Early investment in platformization may pay dividends down the road. There is the possibility to avoid tech debt here if one can create completely product agnostic business platforms and avoid arbitrary product logic in core platform services. It might make sense to adopt extensions at this point to accomplish that goal.

Given that the number of microservices is likely still quite low, it may not make sense to cluster them together. However, it's worth noting here that a domain in the context of Uber's DOMA implementation can contain a single service, so it may still be useful to think in a "domain-oriented" way.

Large

Larger engineering organizations may have hundreds of engineers and microservices and several dependencies. At this point DOMA reaches its full usefulness. There will likely be obvious clusters of microservices that can be easily grouped together into domains with a gateway in front of them. Legacy services often begin to need to be refactored or rewritten and then migrated, which means that gateways will soon begin to provide value in terms of ease of migration if they are already in place.

Clear hierarchy will also become increasingly important with some services operating as "product" services for particular features or grouping of features, and other services will increasingly support multiple products and be thought of as "platforms." It's critical at this stage to keep arbitrary product logic decoupled from platforms, so as to avoid a heavy operational burden on platform teams as well as system-wide instability.

Final Thoughts

We are still actively evolving DOMA as more and more teams at Uber come to adopt it. The critical insight of DOMA is that a microservice architecture is really just one, large, distributed program and you can apply the same principles to its evolution that you would apply to any piece of software. DOMA is simply an approach for thinking about these principles in practice. We hope others find it useful and we look forward to feedback!

DOMA itself was the result of a cross-functional effort, which involved nearly 60 engineers across every org at Uber. Some particular acknowledgements, for people who invested heavily

into this effort over the last 2 years...

Alex Zylman, Alexandre Wilhelm, Allen Lu, Ankit Srivastava, Anthony Tran, Anupam Dikshit, Anurag Biyani, Daniel Wolf, Davide D'Agostino, Deepti Chedda, Dmitriy Bryndin, Gaurav Tungatkar, Jacob Greenleaf, Jaikumar Ganesh, Jennie Ngyuen, Joe McCabe, Joshua Shinavier, Julia Law, Kusha Kapoor, Linda Fu, Madan Thangavelu, Nimish Sheth, Parth Shah, Shawn Burke, Simon Newton, Steve Sherwood, Uday Kiran Medisetty, and Waleed Kadous

Acknowledgements:

This work brings multiple existing design patterns in the industry to solve problems at Uber while also suggesting new patterns like extensions. We are thankful to the industry for the work on them. We are also thankful to the engineers at Linkedin who worked on Superblocks, who spoke to us about their experiences.

Adam Gluck

Adam Gluck is a Sr. Software Engineer II at Uber. He spent his first 3.5 years at Uber fleshing out our Driver Platform team and helping to scale our driver product. More recently, he's been a part of Uber's engineering strategy team, focused on high level system architecture and Uber-wide platformization efforts.

Posted by Adam Gluck

Category:
Engineering