**Good morning everyone.**

Today I’ll be taking you through our project on **Microservices Refactoring**, where we focus on breaking down a monolithic architecture into a microservices-based approach, using our Shopify-like platform as an example.

**Why break the monolith?**

To start with — **Shopify, like many modern platforms, initially began as a monolithic application**.  
This made a lot of sense in the early days — it was straightforward to build, quick to deploy, and all the features like vendor catalogs, cart, payments, and user profiles lived together in one big codebase.

But as the platform grew — with more vendors joining, more customers shopping, and new features rolling out — this monolithic approach started showing serious cracks.

We faced a few key problems:

* **Scaling inefficiencies:** If just the cart was under heavy load during a festival sale, we still had to scale the entire application. This meant wasting compute and money on services that weren’t under pressure.
* **Risky deployments:** Even a tiny bug in the payment module could unexpectedly take down vendor dashboards or user profiles, because everything was tightly coupled.
* **Innovation bottlenecks:** Trying out a new tech stack — say for personalized recommendations — was almost impossible without rewriting large chunks of the codebase.
* And finally, **team slowdowns:** With everyone working on the same giant codebase, we ran into merge conflicts, painfully long test cycles, and slow releases.

**Benefits of moving to microservices**

By breaking the monolith into **microservices**, we aimed to solve these challenges.

Some of the big benefits we gained are:

* **Independent scaling:** We can now scale just the cart service during a sale, instead of the whole app.
* **Faster, safer deployments:** We can push a payment fix without redeploying the entire platform.
* **Tech flexibility:** Teams can pick the best tools for their service — maybe Node.js for the product catalog, Spring Boot for orders, or Python Flask for payments.
* **Fault isolation:** If the promotions service crashes, customers can still browse products and check out without issues.
* And it’s much **easier for new team members to onboard**, because they only need to understand a small, focused codebase.

**How we identified our microservices**

So, the next big question was — **what exactly do we break it into?**

We followed three key principles:

1. **Business capabilities:** We looked at what our business actually does — things like managing carts, handling orders, processing payments, managing the product catalog, and running promotions. Each of these naturally became a candidate for a microservice.
2. **Change frequency:** We analyzed our Git history and project boards to see which modules changed most often. Unsurprisingly, the cart, promotions, and catalog had the most activity, so we prioritized them.
3. **Data ownership:** Each microservice owns its own database. For example, the Cart Service manages cart data, the Payments Service handles transaction logs, and so on. This means services can evolve independently without tangled database migrations.

**Our identified microservices**

With this approach, we settled on five core services:

* **Cart Service:** Manages cart items, sessions, and applies promotions.
* **Payments Service:** Handles payment processing and transaction logs securely.
* **Orders Service:** Manages order statuses, shipping, returns, and invoices.
* **Catalog Service:** Takes care of vendor uploads and inventory.
* **Promotions Service:** Runs discounts, campaigns, and validates coupons.

**Containerization strategy**

To run all of this smoothly, we used a **containerization strategy with Docker, Amazon ECR, and ECS**.

* **Dockerization:** Each service is containerized, ensuring it runs the same way on a developer’s laptop, on staging, and in production.
* **Image management:** We push all our Docker images to **Amazon ECR**, so we have a centralized, secure registry.
* **Deployment on ECS:** Services are deployed using **Amazon ECS** with EC2 launch type. This handles scheduling, running, and auto-recovering containers.
* For example, our **Cart Service** scales up automatically when CPU exceeds 60%, while the **Catalog Service** scales based on HTTP throughput.

**Now that we’ve covered why we chose microservices and how we identified them, I’ll give you a quick overview of how we’ve designed this on AWS, before my friend dives deeper into the architecture.**

**First, the frontend:**  
We’re hosting our frontend static assets — like HTML, CSS, and JavaScript — on **Amazon S3**, which acts as a durable object store. This is served to users via **CloudFront**, a global Content Delivery Network, to ensure fast load times anywhere in the world. So whether a customer is in India, the US, or Europe, they get a quick and responsive shopping experience.

**Next, the API Gateway:**  
This acts as the **single entry point** for all client requests. Whether it’s adding an item to the cart, processing a payment, or fetching a product catalog, the **API Gateway routes these calls to the appropriate microservice**.  
It also gives us features like throttling, authentication checks, and request transformations.

**Finally, service mapping:**  
At the core of our architecture is the mapping of each microservice to the right compute and data layer.  
For instance:

* The **Cart Service** might run on ECS with DynamoDB for low-latency cart data.
* The **Orders Service** could use ECS with RDS for structured order records.
* The **Payments Service** integrates with SQS to decouple transaction processing.

This design gives us flexibility in choosing the right database and compute for each microservice, improving performance and maintainability.

**Once we adopted AWS to host our microservices, we gained access to a rich ecosystem of auxiliary services that really strengthened our deployment, security, and migration strategy.**

**First, on security and isolation:**  
We’ve implemented **IAM roles** to make sure each microservice only has access to exactly what it needs — following the principle of least privilege. Alongside this, we use **AWS KMS** for encrypting our data and managing secrets, ensuring sensitive information stays protected.

**Next, container deployment:**  
Each of our microservices is containerized and deployed on **Amazon ECS**. ECS not only helps us manage and orchestrate these containers, but also enables **auto-scaling** based on metrics like CPU or memory, so we’re always using resources efficiently.

**Then for logging and monitoring:**  
We rely on **Amazon CloudWatch** to track logs and metrics across our services, and **AWS X-Ray** to trace requests end-to-end. This helps us quickly debug performance bottlenecks and maintain high reliability.

**Now, about migration:**  
We’re using **AWS Migration Hub Refactor Spaces**, which is basically our smart traffic manager. It lets us route some requests to the old monolith and some to our new microservices. This ties into the **Strangler Fig pattern** — where we slowly phase out the monolith by cutting over individual functionalities to microservices, minimizing risk and downtime.

✅ **Final Benefits**

* **Cost efficiency:** Scale only what’s needed, keeping costs aligned with actual demand.
* **Faster deployments:** Roll out features and fixes quickly without touching unrelated parts of the system.
* **Increased resilience:** Failures stay isolated, so one service going down doesn’t take the whole platform with it.
* **Independent teams:** Teams work on and deploy their own services without stepping on each other’s toes.