SKILUP SM DAYS

by: DevOps INSTITUTE

Cracking the Case: How to Migrate from Monoliths to Microservices

Andrew Chee
Senior Solutions Engineer
Lightstep
@LightstepHQ
andrew@lightstep.com

Hi, I’m Andrew

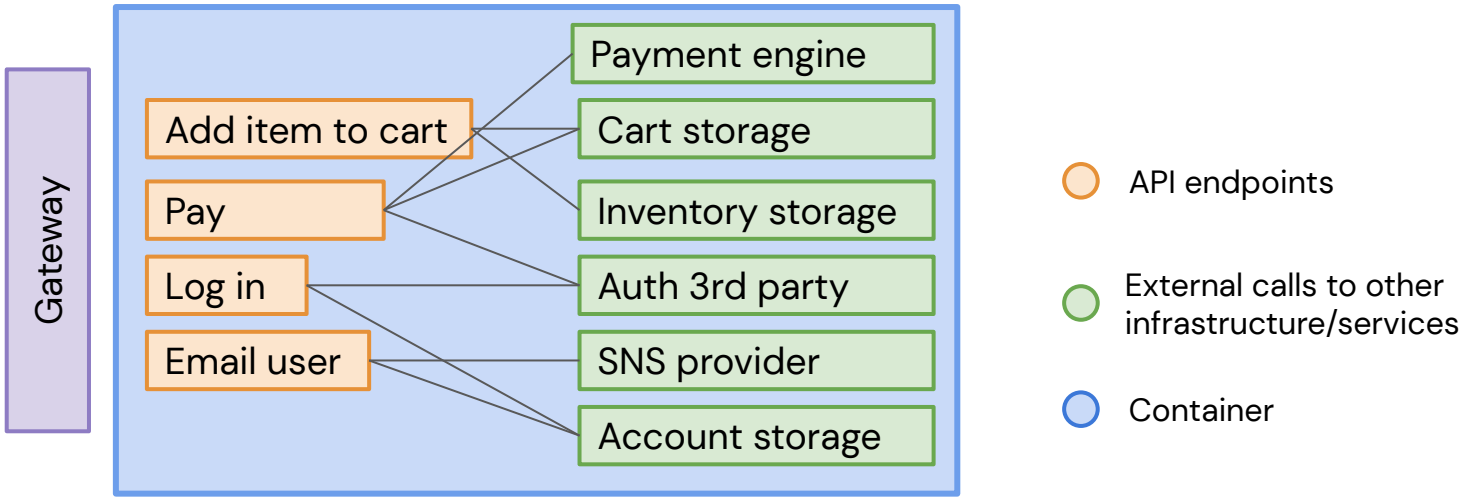
a Solutions Engineer at Lightstep

Quick Agenda

1. Walk through general service migration strategy (with an example architecture) (15 min)
2. How to use observability to de-risk a migration (10 min)
3. What does modern observability for microservices look like? (5 min)

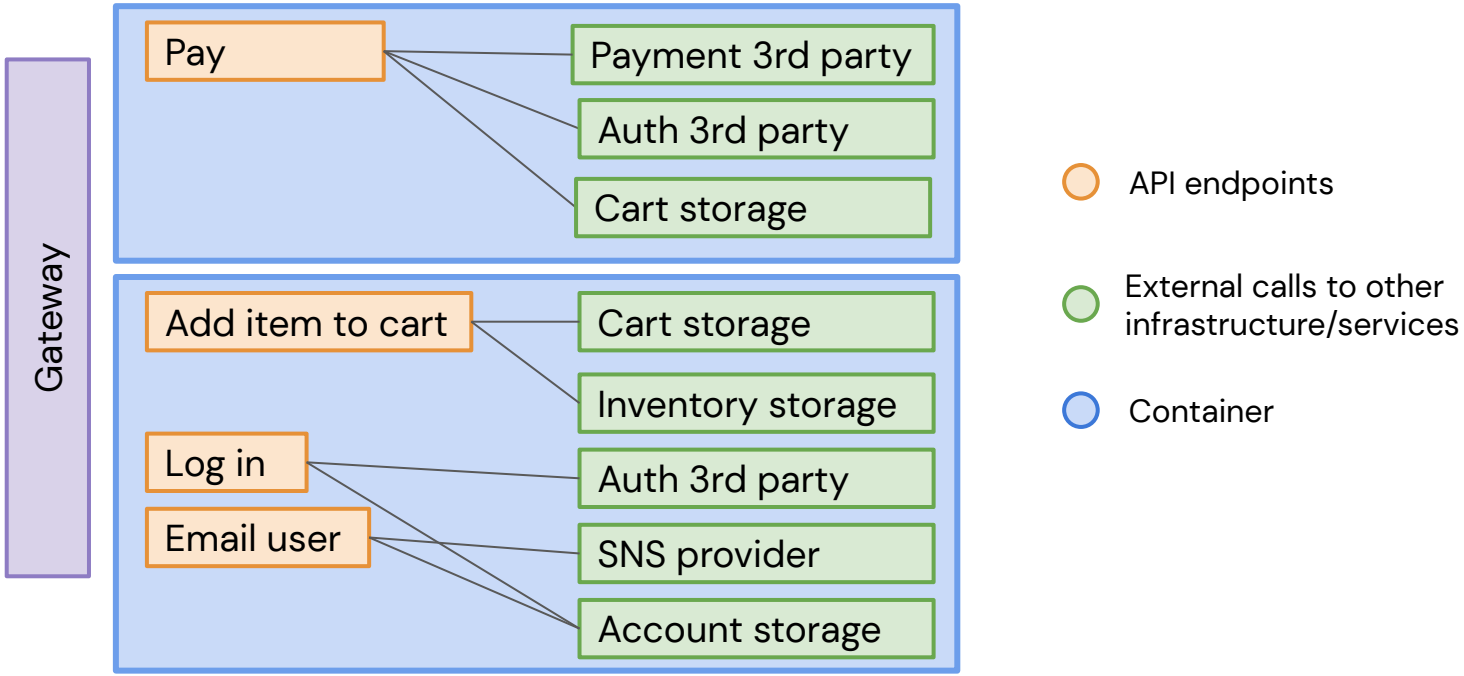
Our Example Monolith

→Todo: Split out payments into its own microservice, integrate it with a new 3rd party



Our Goal State

→ Payments is split out from the monolith, with no downtime to upstream services and end-users



Migration Exercise

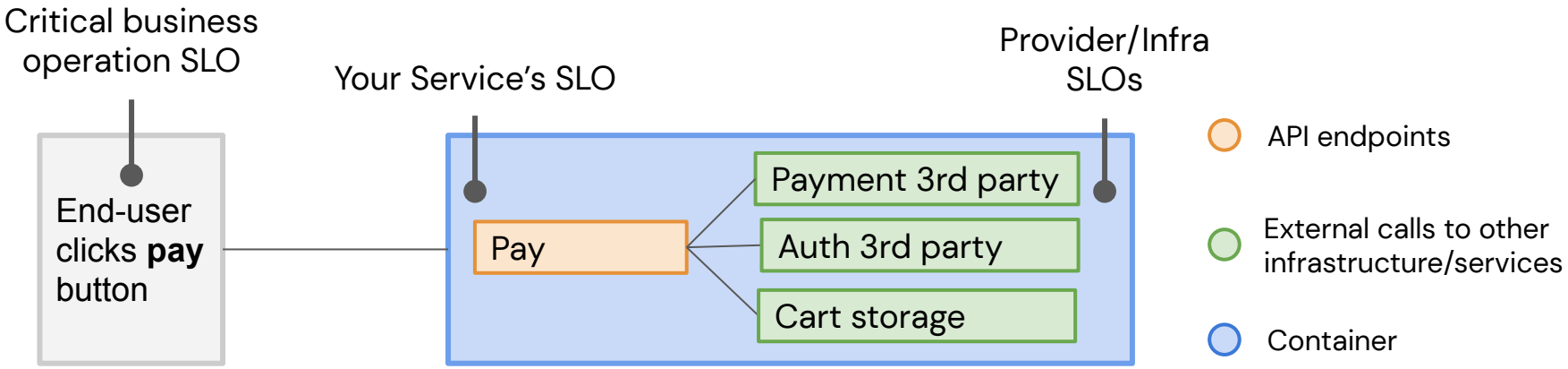
- Scenario: Current payments are handled in the monolith, we want something horizontally scalable, integrated with a new 3rd party, operating asynchronously from the rest of the stack
- **Tip: Define must-have solutions upfront. Start thinking about SLOs upfront.**
- Example must-haves: More scalability, more reliability, integrate the new 3rd party, zero downtime

Migration Strategy: SLOs

- Everyone has opinions on SLOs, but no one knows the best way to define them. Here is some general guidance...
- **Tip: Define your internal service SLOs based on:**
  a. **Services upstream that rely on your services**
  b. **Services downstream that you rely on** (protect your team from blame)
- **Tip: Define executive SLOs based on critical end-user business operations** (what will cause loss of time/money/users if it breaks?)

You can use Observability systems that leverage tracing and metrics to measure and define end-to-end SLO's.

Migration Strategy: SLOs



Example provider SLA:: Alert when the 3rd party payment service error rate > 0.1%

Example internal SLA: Alert when /pay endpoint's p99 > 5 sec

Example critical SLA: Alert when clicking Pay button on a validated payment option's error rate > 1%

Migration Strategy: Operations

- Make sure containerization, configuration management, permissions management and CICD is set up in a repeatable way for new services. Add any security must-haves and do rough cost analysis upfront.
- **Tip:** This may seem obvious, but **use a familiar programming language for the service unless there's a good reason not to.**

Q: When should you use serverless/lambda?

Depends on how familiar your team is with the tech and how complex the business logic is. If it takes > 30 seconds to process certain requests, then consider using short-lived container (ex. Fargate) or a microservice.

Migration Strategy: API Contract

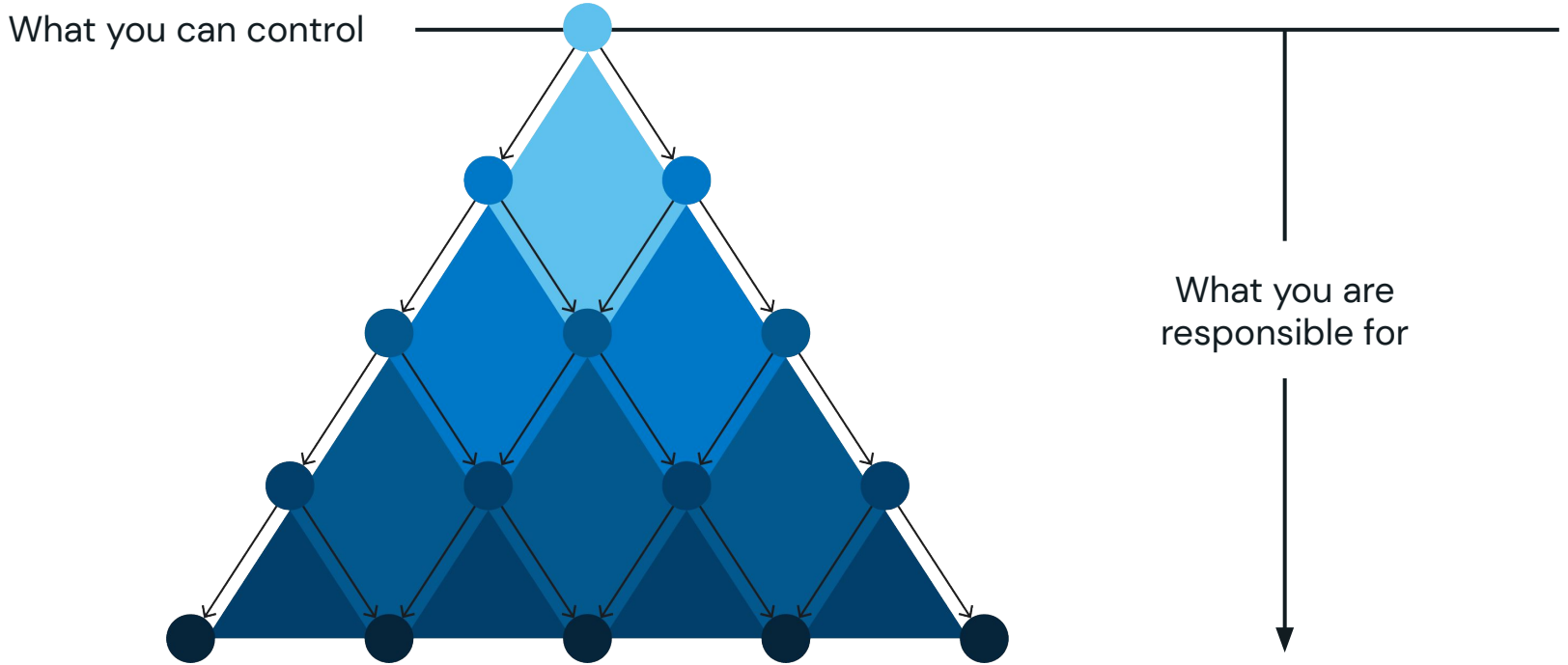
- How is the new service going to be exposed to the outside world, what kind of requests does it need to handle to support any new functionality?
- Understand and diagram *existing* behavior that will be migrated out of the monolith. Understand current performance limitations, and add these to the must-have list (possibly attach SLO's to enforce)
- **Tip: Use something like Swagger, if you have time and want to enforce API stability.**

You can use Observability systems that leverage tracing to understand current performance limitations and bottlenecks.

Migration Strategy: Dataflow

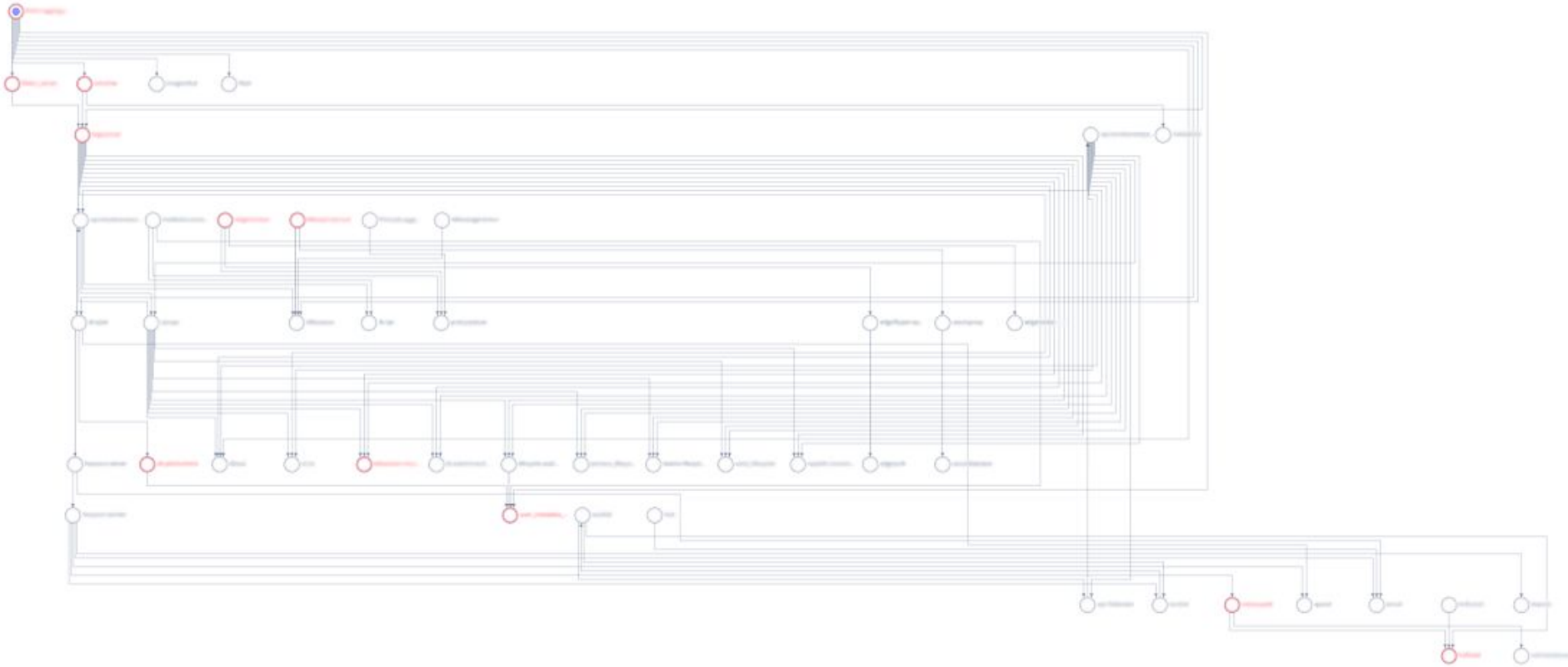
- Think about data models: In our example, what is the new logic and data model for the 3rd party service we are integrating with? In general, what external infrastructure are we relying on in our new service?
- **Tip: Brainstorm risks upfront.** De-risking activity could fall under benchmarking, throughput analysis of current system, management plans and playbooks for any new external infrastructure or services
- **Tip: Create a plan for how to migrate the old service to the new one. Plan for the plan to fail.** Will it be a % of traffic? One endpoint at a time? When will the migration be considered confidently “done”? How will the service rollback?

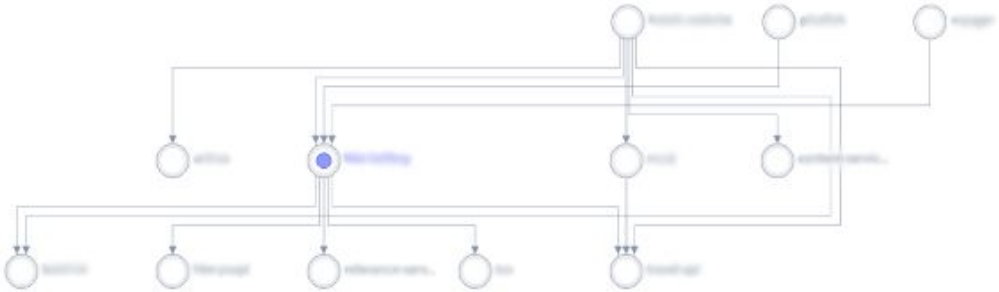
The Microservice Effect

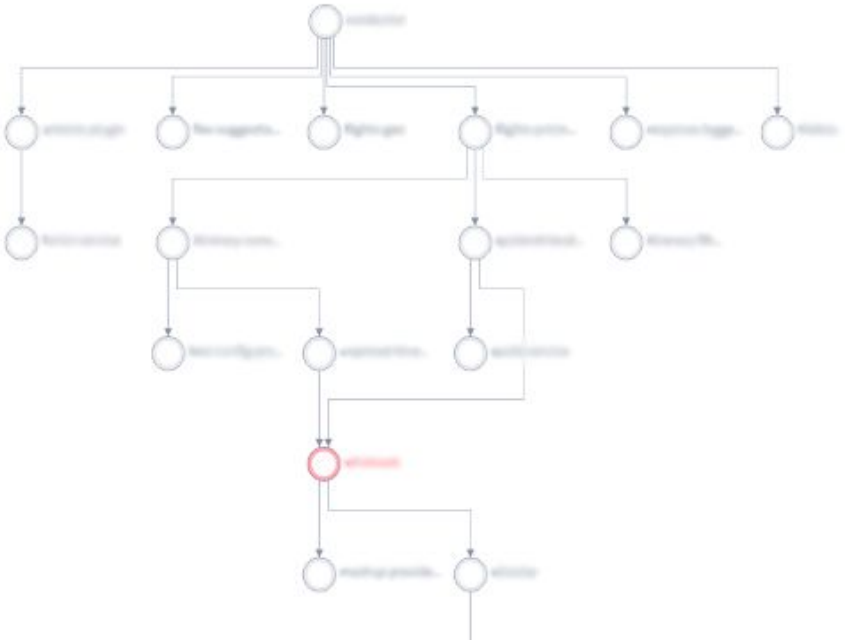


Observability Challenges

- **Microservices introduce complexity.** We're going from something that's relatively straightforward to reason about (mostly due to coupling) into decoupled, distributed parts
- **Microservices introduce more points of failure** by adding additional service-to-service boundaries and spreading out calls to infrastructure and external services.
- Examples of real "deep" systems from our platform →







De-risking for Code Complexity

Leverage Observability for:

- Identifying which code paths and dependencies in the monolith are going to be migrated to the microservice
- Verifying when those code paths are able to be deprecated and no longer serving traffic
- Verifying during deployments of the new service (and old monolith) that functionality isn't regressing unexpectedly

De-risking for Performance Regressions

Leverage Observability for:

- Identifying bottlenecks, both in the monolith and in the new microservice
- Collecting throughput, error rate and response time requirements that the new microservice needs to support
- Collecting throughput, error rate and response time requirements for new external infrastructure and 3rd party services

De-risking for External Dependencies

Leverage Observability for:

- Enabling developers to see how the new service code is performing in real time as it's being built. As soon as a developer creates code calling a new external service, they should be able to
    a. Verify the external service is responding
    b. Collect performance metrics on this external service (such as status codes, response time, error rate)
- Wrapping the new service in easily debuggable and understandable SLOs, to give everyone a clear picture on the state of the world both during and after completing the code migration

THANK YOU!

Meet me in the Network Chat Lounge for questions

SKILUP DAYS