HAMBURG, GERMANY | 20th May 2026

AWS SUMMIT

MAM303

Circuit Breaker, Saga & Strangler Fig - Patterns for transformation

Adrian Begg

he/him

Senior Solutions Architect

Amazon Web Services

Patrick Meiler

he/him

Solutions Architect

Amazon Web Services

Robert Hanuschke

he/him

Senior Solutions Architect

Amazon Web Services

All photos by Dirk Fröhner

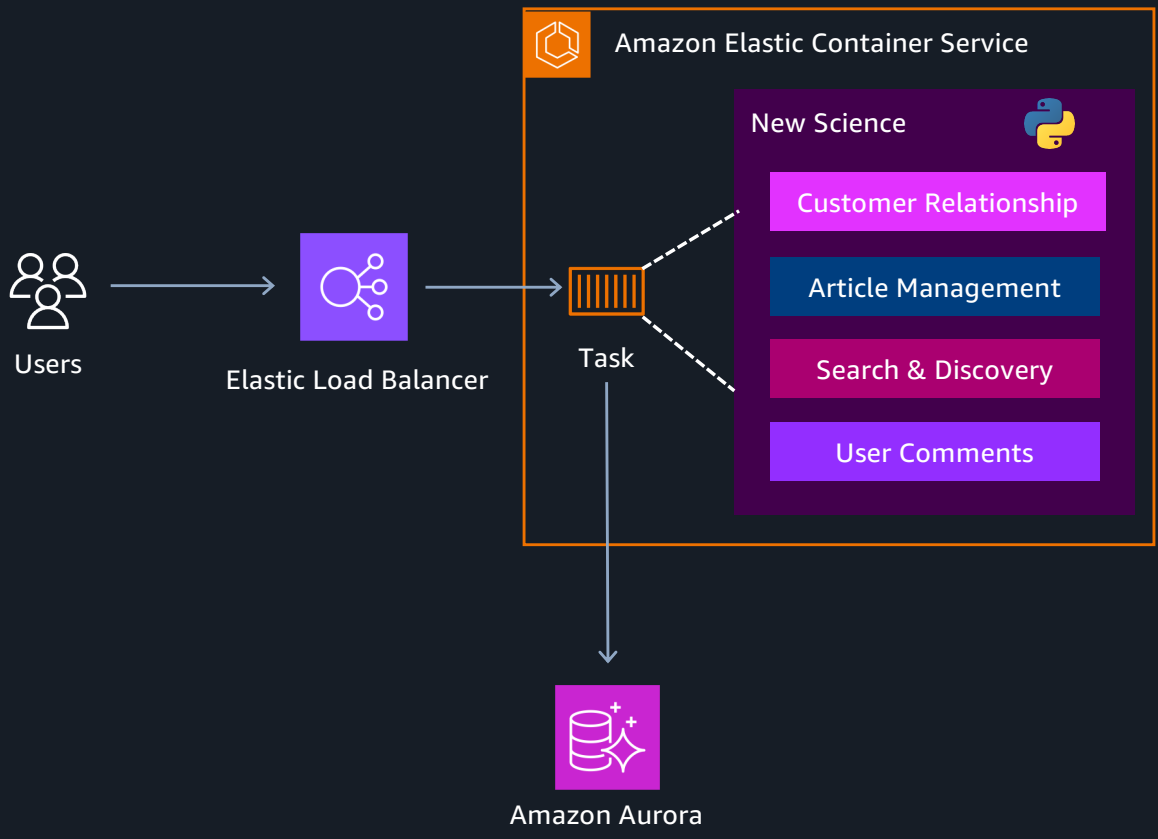
No architecture decision you take comes without trade-offs.

Your job as software architect is to identify the least painful option on the table.

OUR APPLICATION

AnyCompany New Science

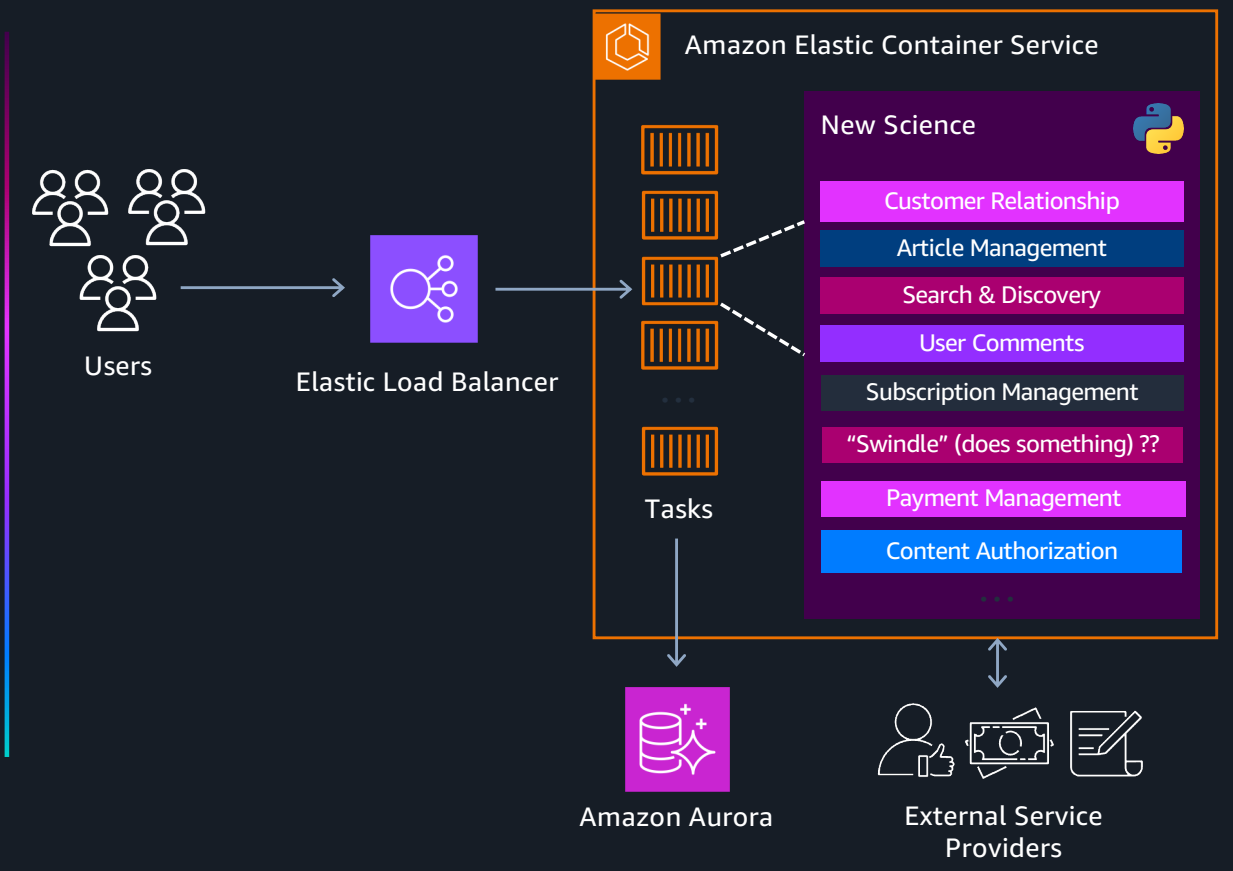
How it all began ...



OUR APPLICATION

AnyCompany New Science

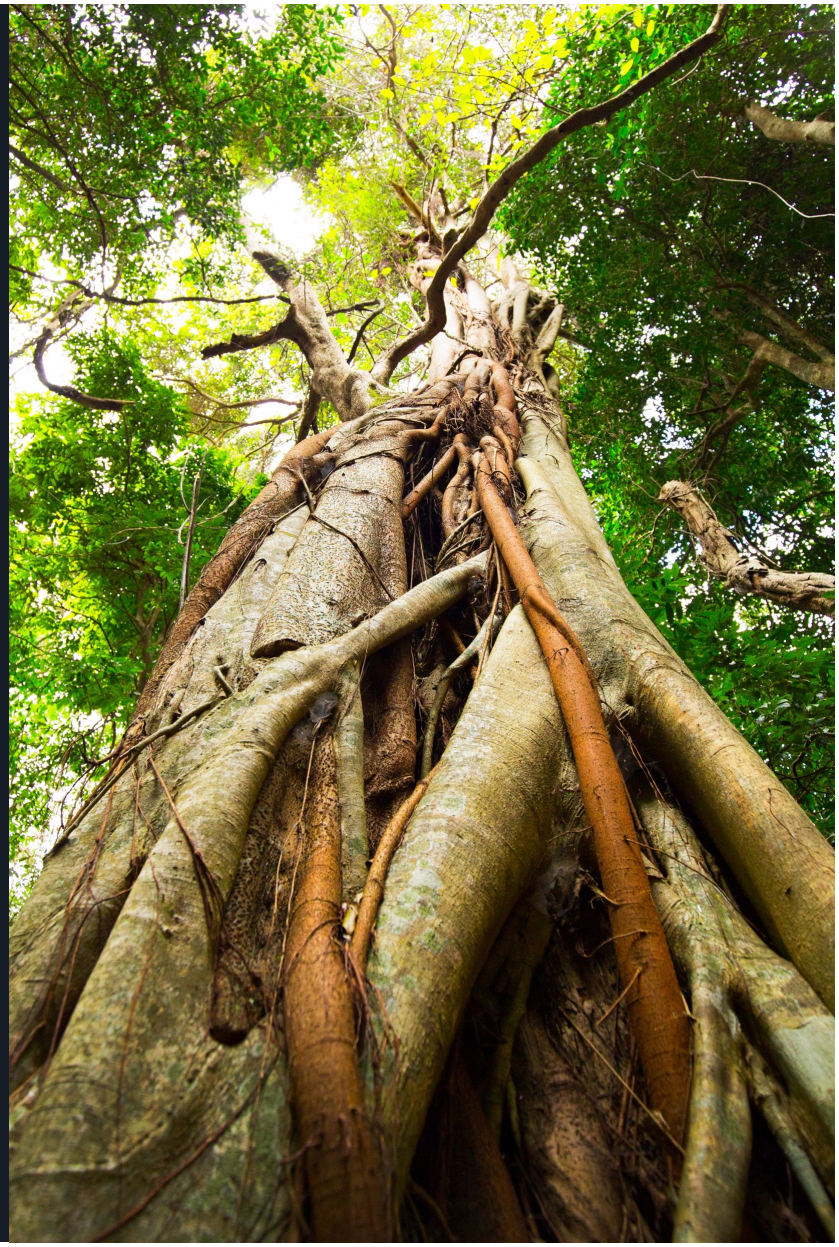
5 years and many feature requests later …



Strangler fig

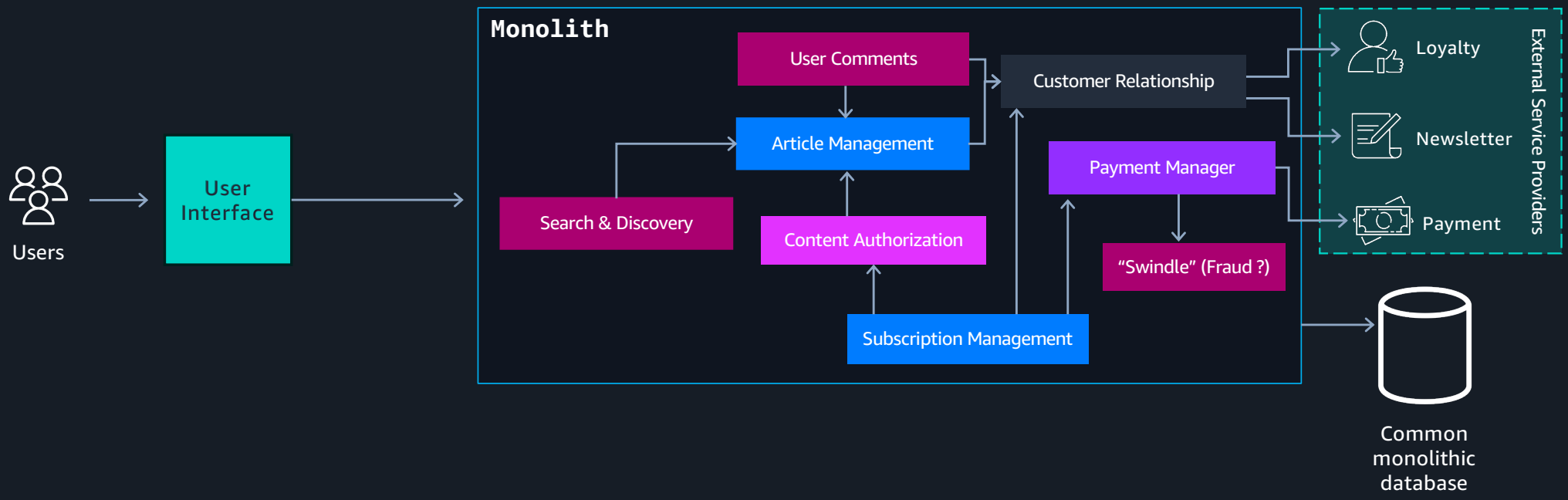
Strangler Fig (with Façade) Pattern

Gradually create a new system around the edges of the old, letting it grow slowly until the old system is strangled



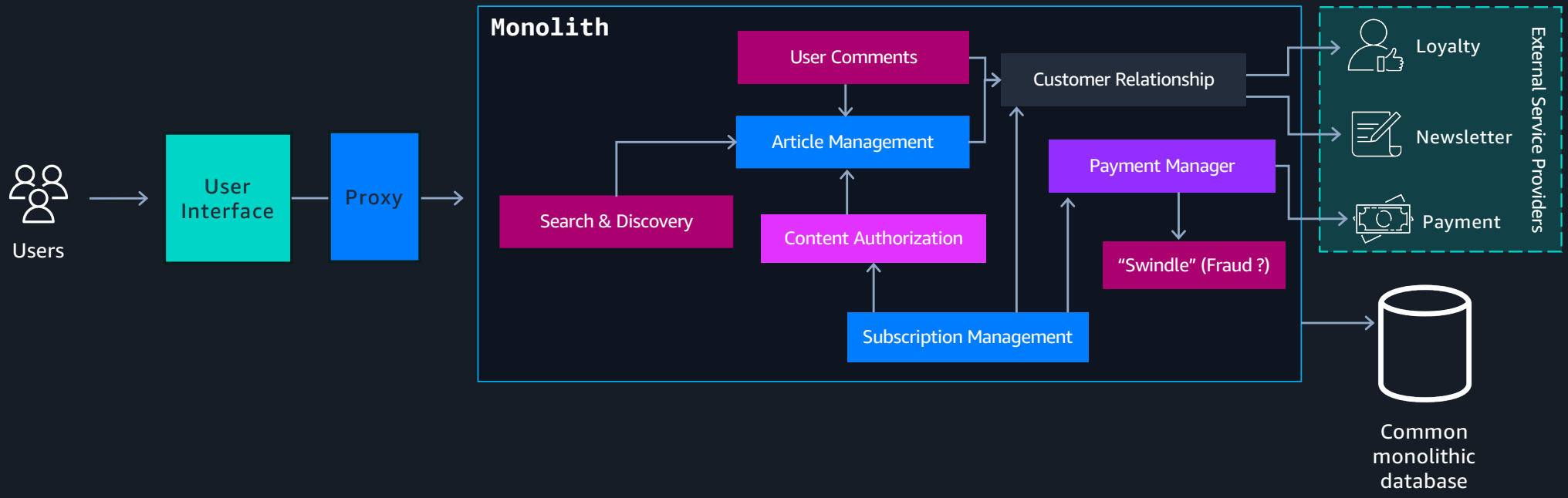
Strangling the Monolith

Our starting point, the monolithic application



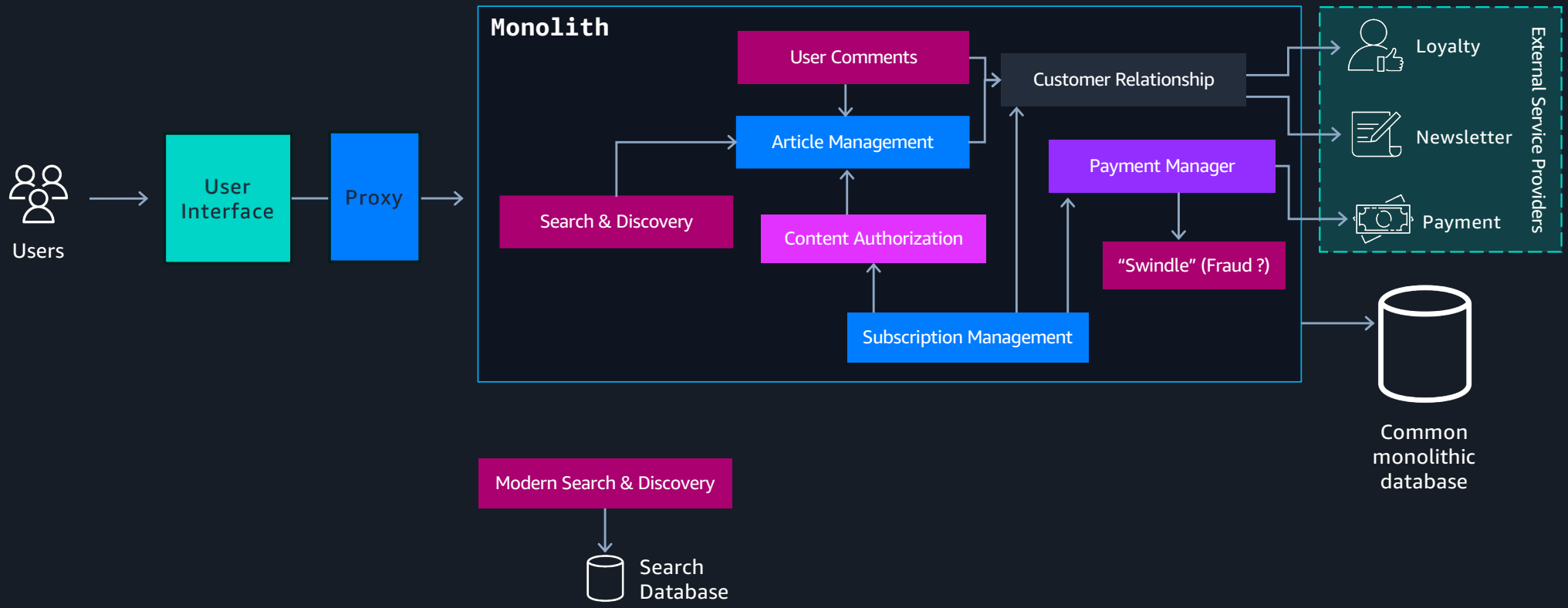
Strangling the Monolith

Introduce façade layer to proxy user requests



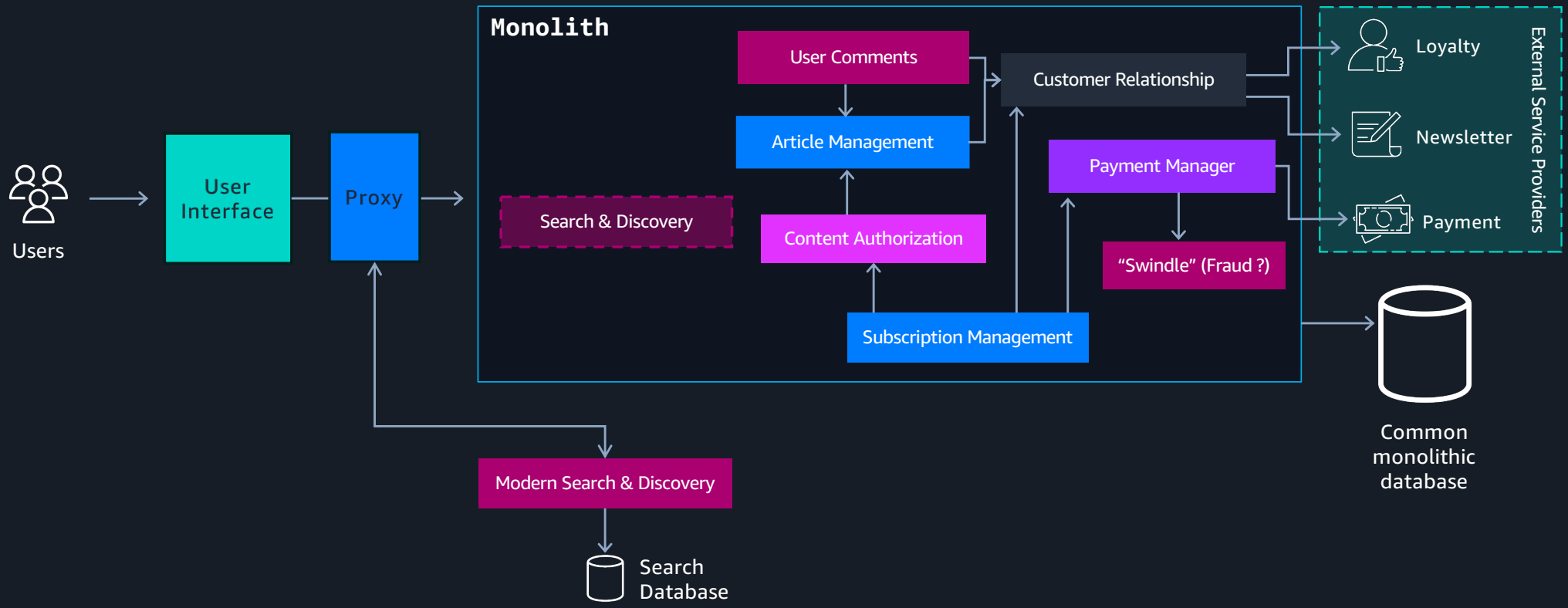
Strangling the Monolith

Deploy new modernized service and redirect traffic with proxy



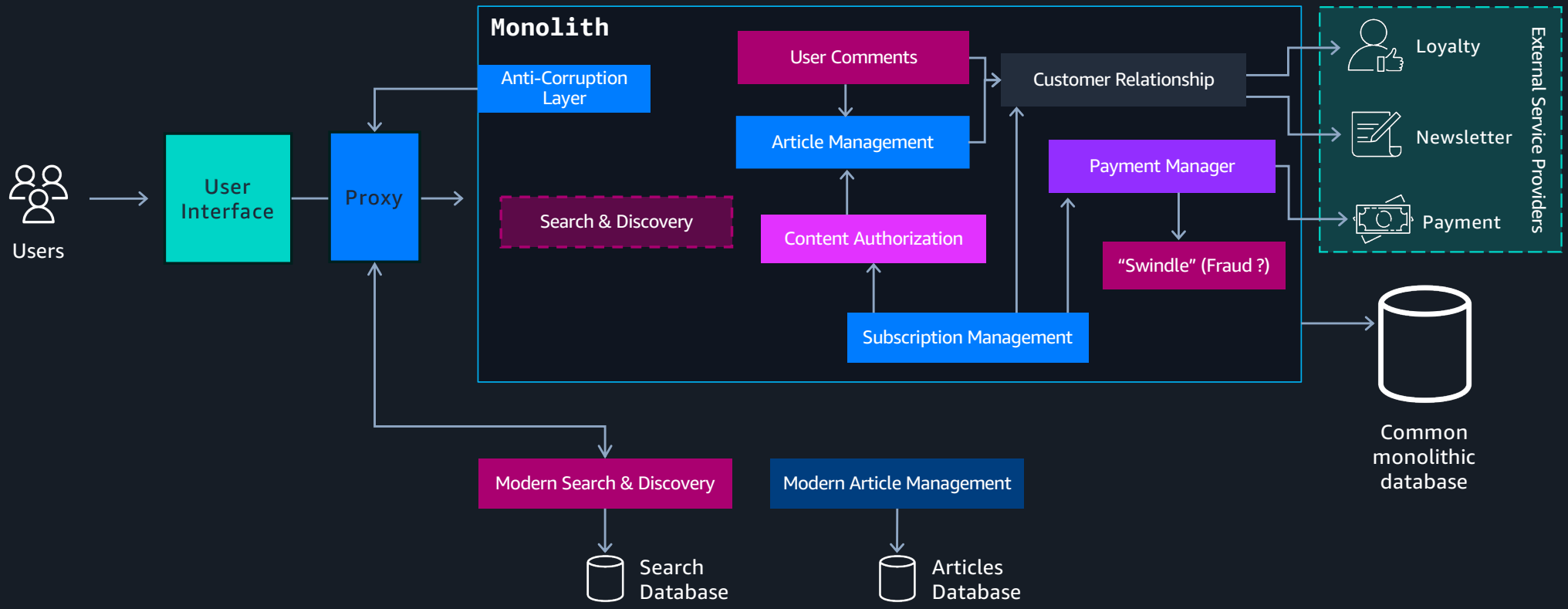
Strangling the Monolith

Deploy new modernized service and redirect traffic with proxy



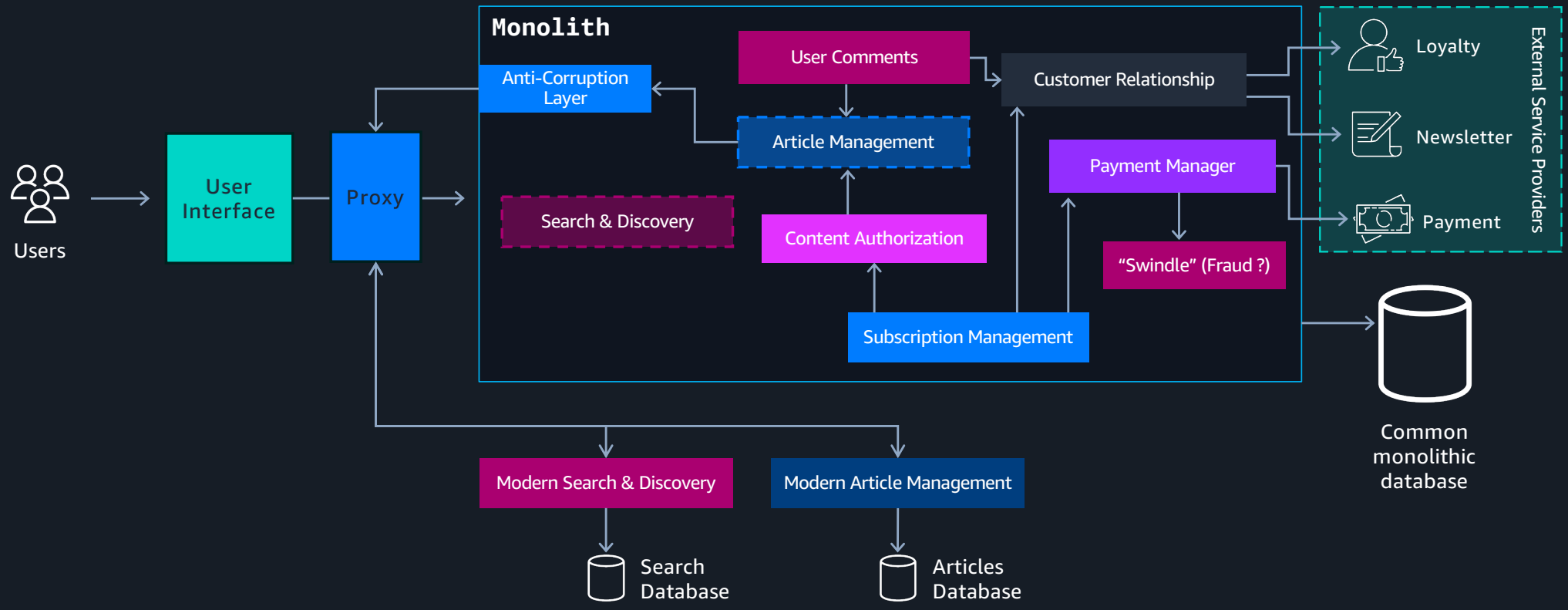
Strangling the Monolith

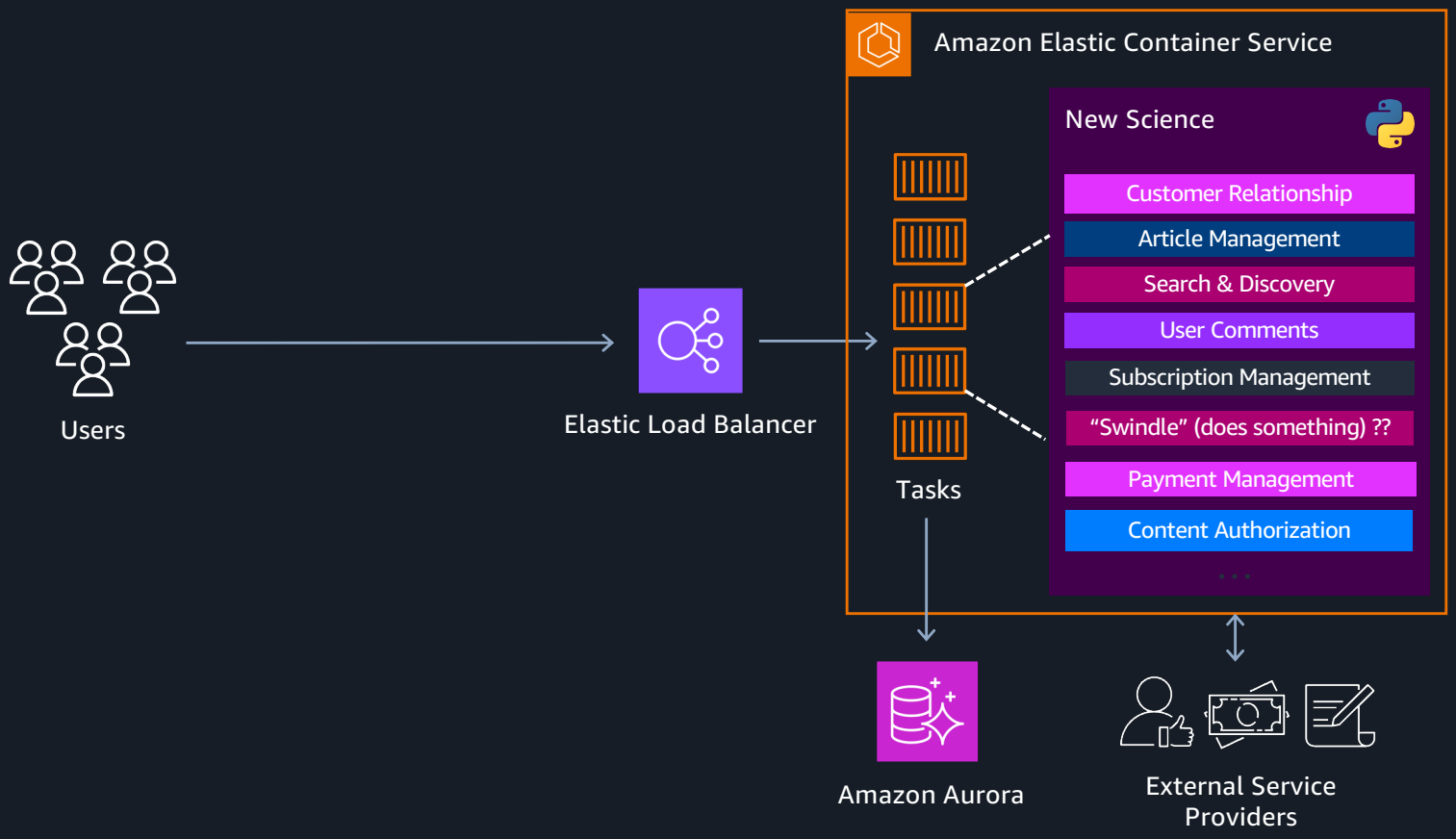
Implementing a modern article management service and an anti-corruption layer (ACL) to our monolith

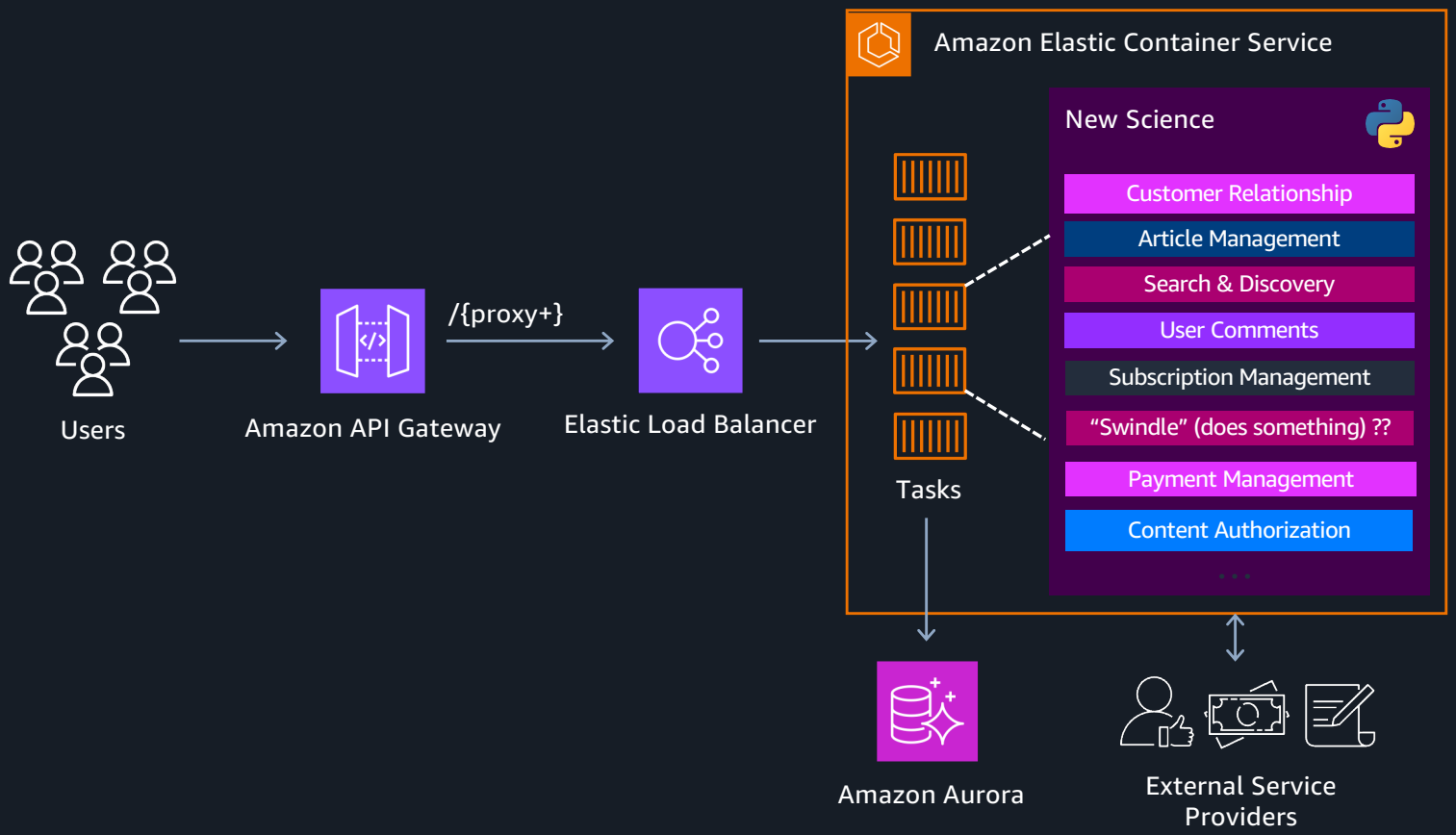


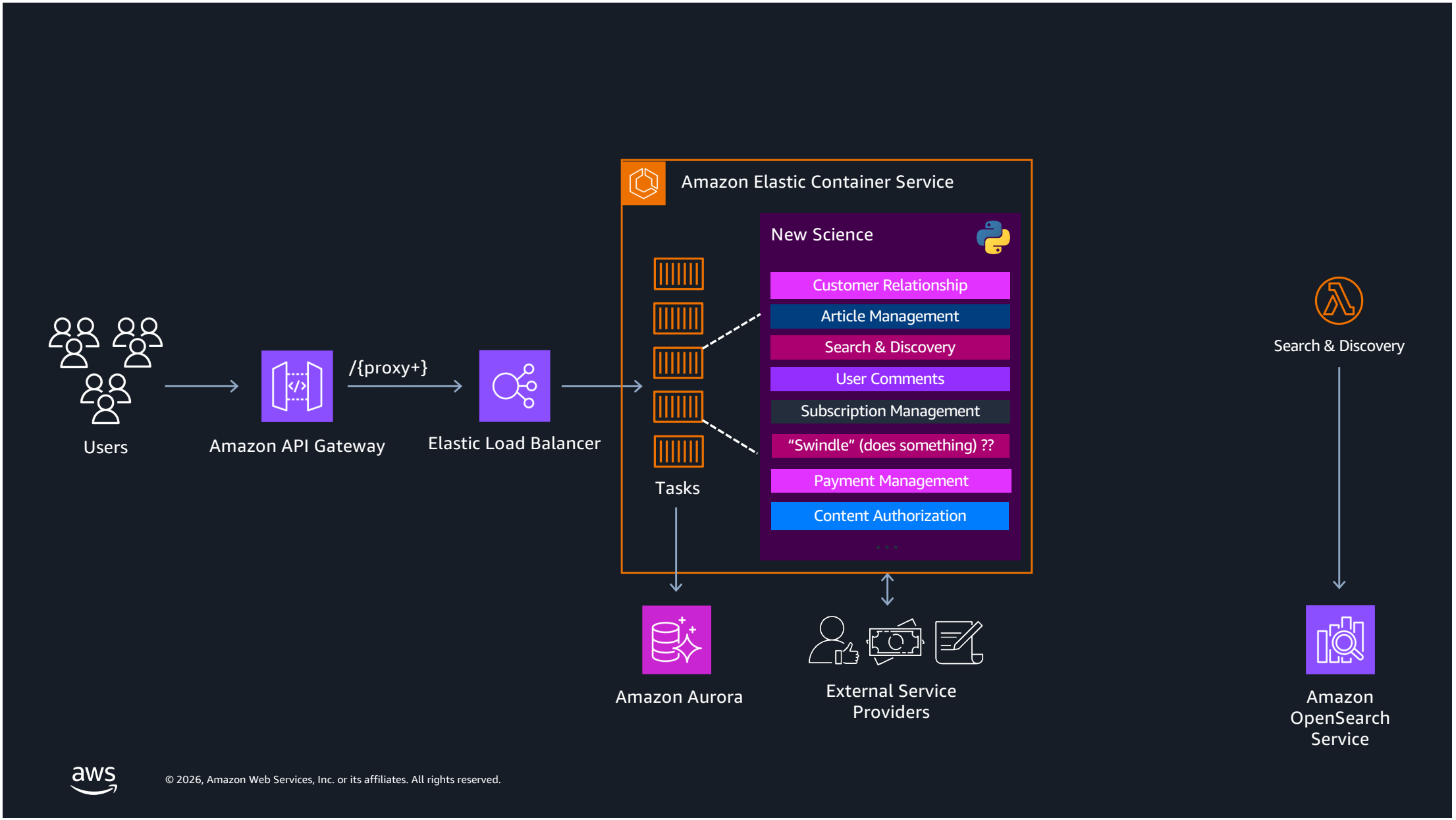
Strangling the Monolith

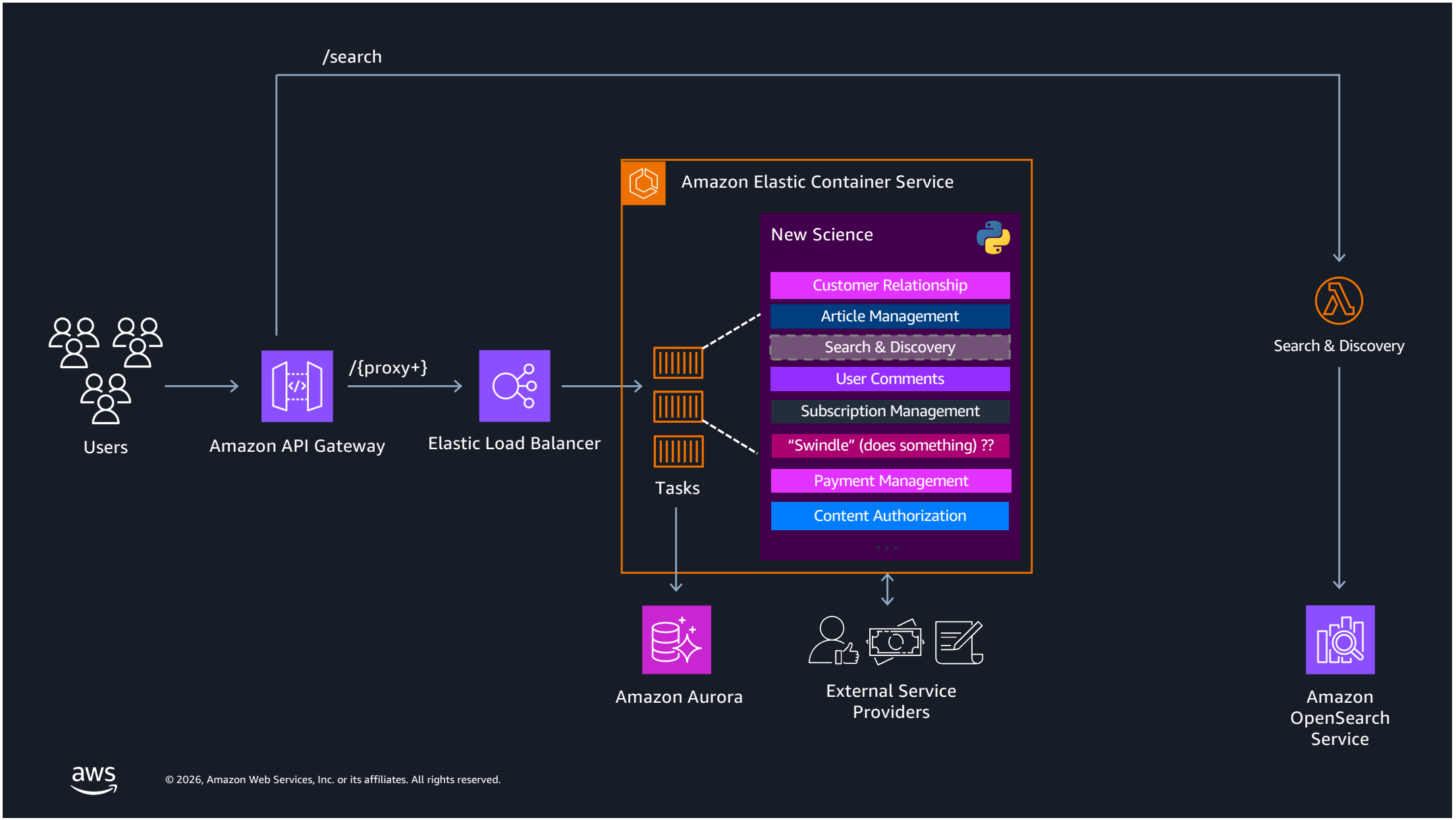
Cutover, Iterate, prioritize and incrementally modernize based on benefits or solving actual pain points

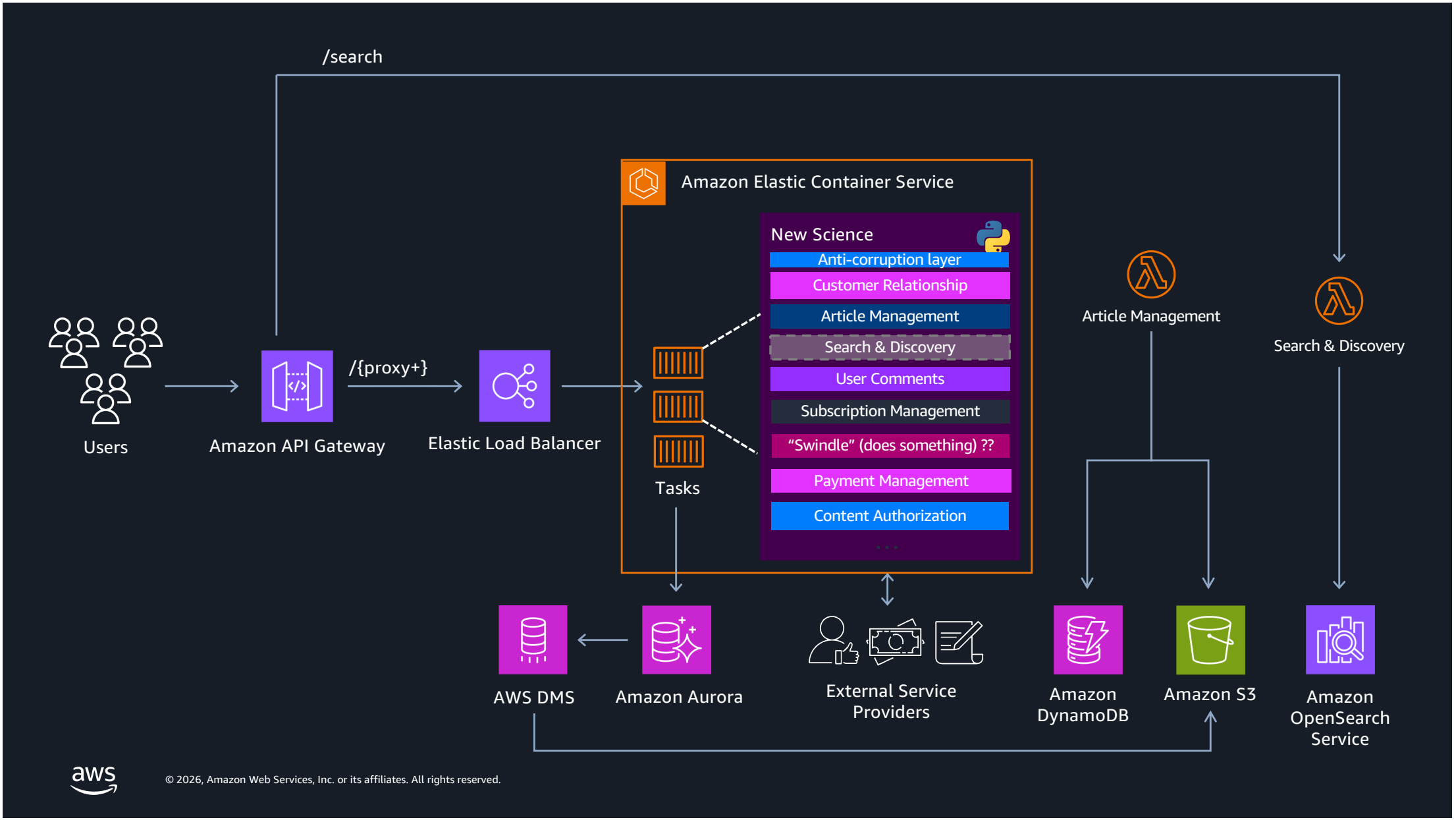


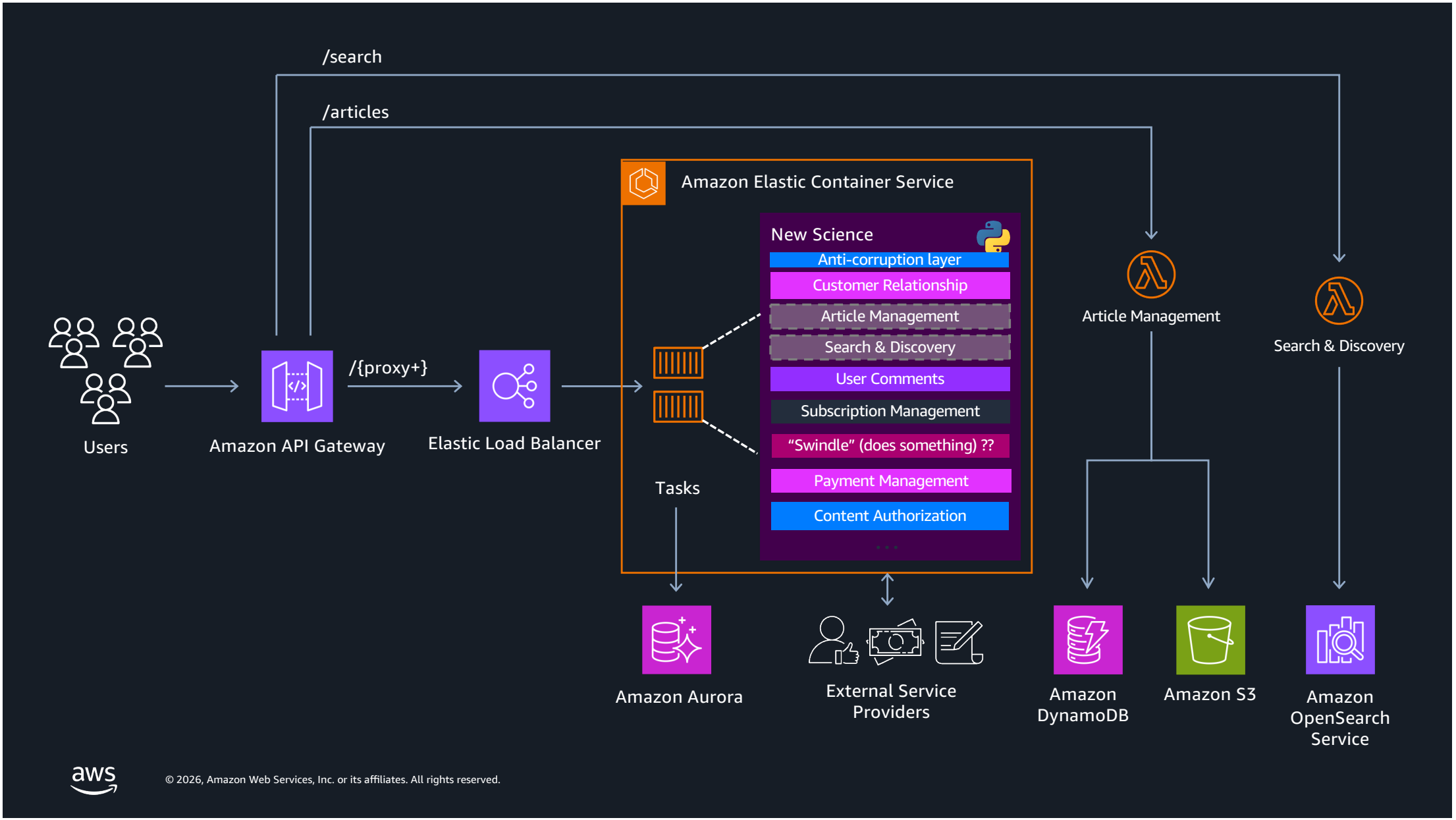












Strangling the monolith

Think big but start small, gradually build new parts of the system around the edges, hollowing out the monolith

Managing interaction between systems:

Saga Patterns

Divide and conquer

The monolith
architectural style
makes it easier to neglect
your clean architecture
over time.

The independent systems
architectural style
makes it easier to retain
your clean architecture
over time.

Of course, not for free.

Use case: Subscribe to premium tier

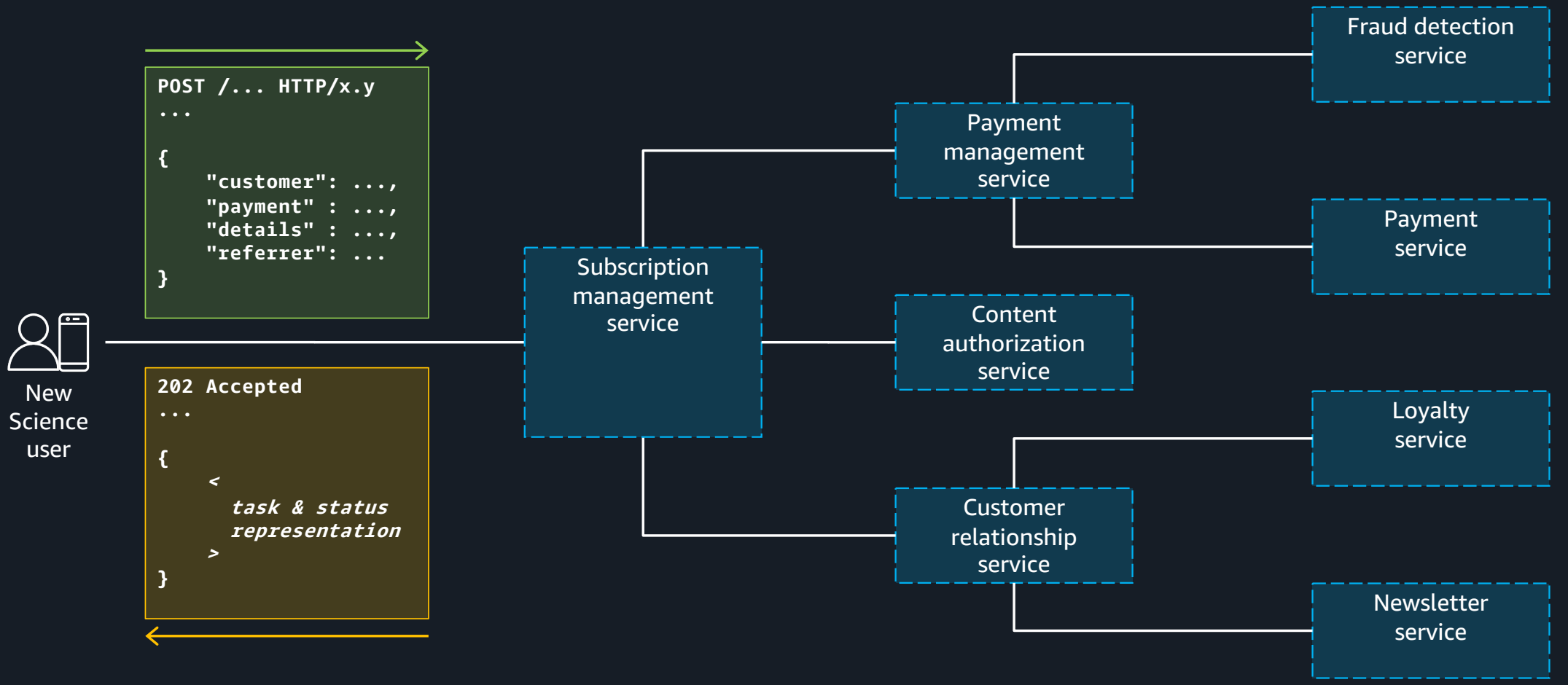
A lot of steps to fulfil in order to subscribe a customer to premium tier (incomplete):

- Customer sends subscription request to subscription management
- Subscription management invokes payment management
- Payment management invokes fraud detection
- Payment management invokes payment collection
- Subscription management invokes content authorization
- Subscription management invokes customer relationship management
- Customer relationship management invokes loyalty program
- Customer relationship management invokes newsletter management

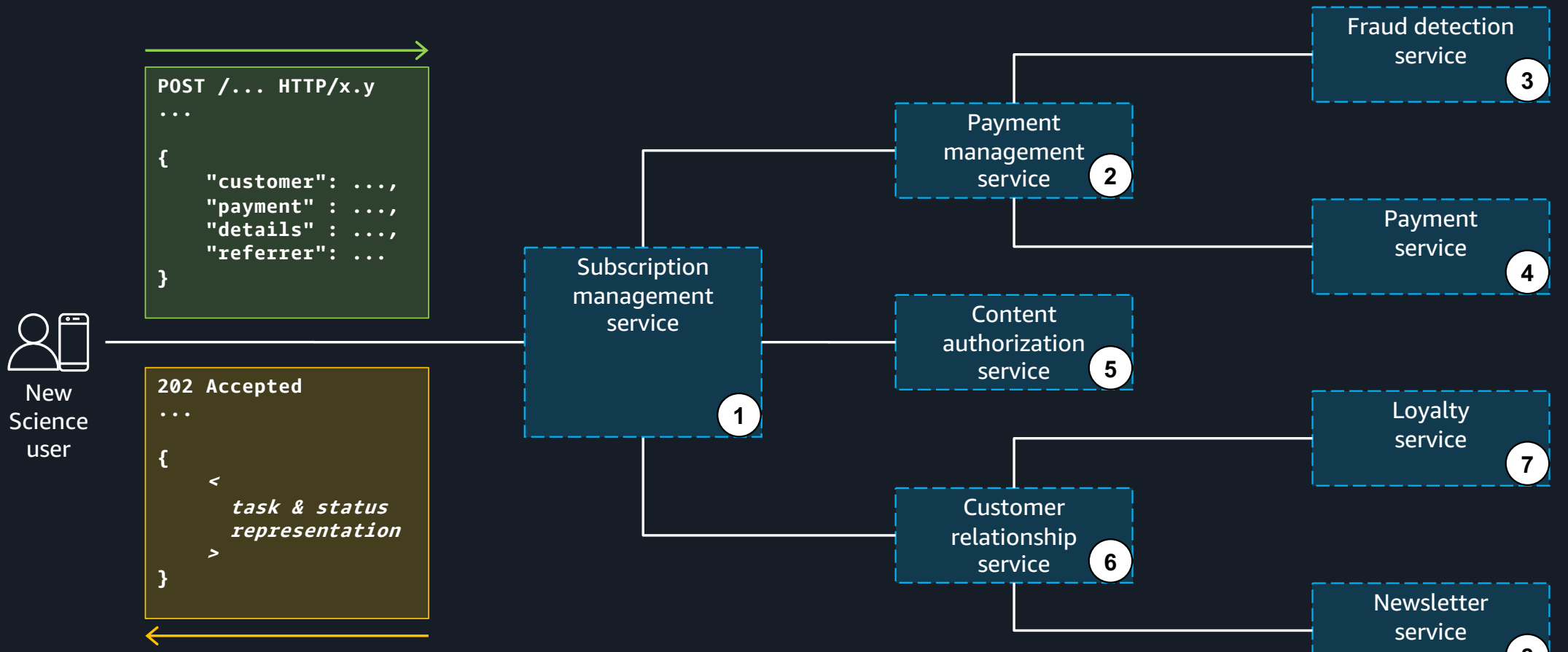
Business decisions (not tech decisions) how to handle failures (incomplete):

- Abort entire request
- Retry a failing step, maybe do it later
- Report back to user either way

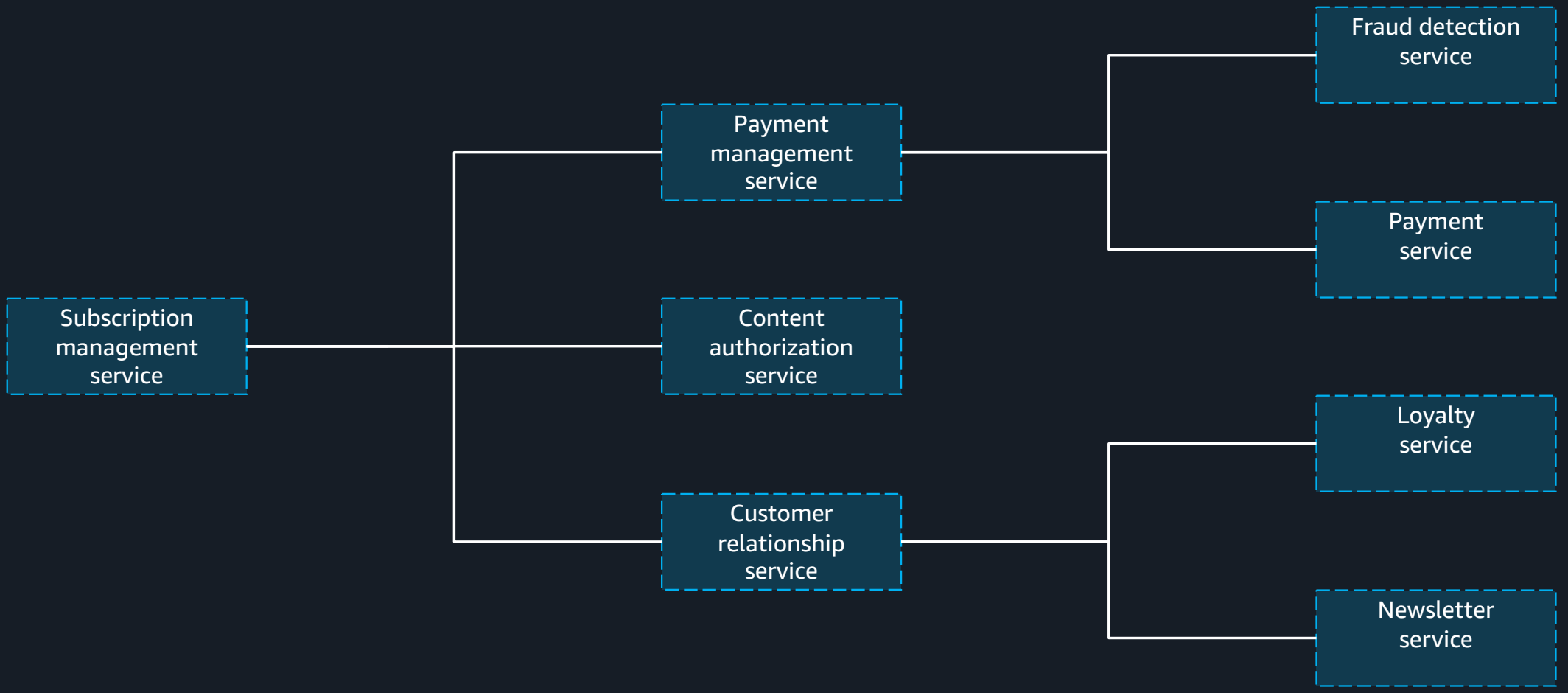
Use case: Subscribe to premium tier



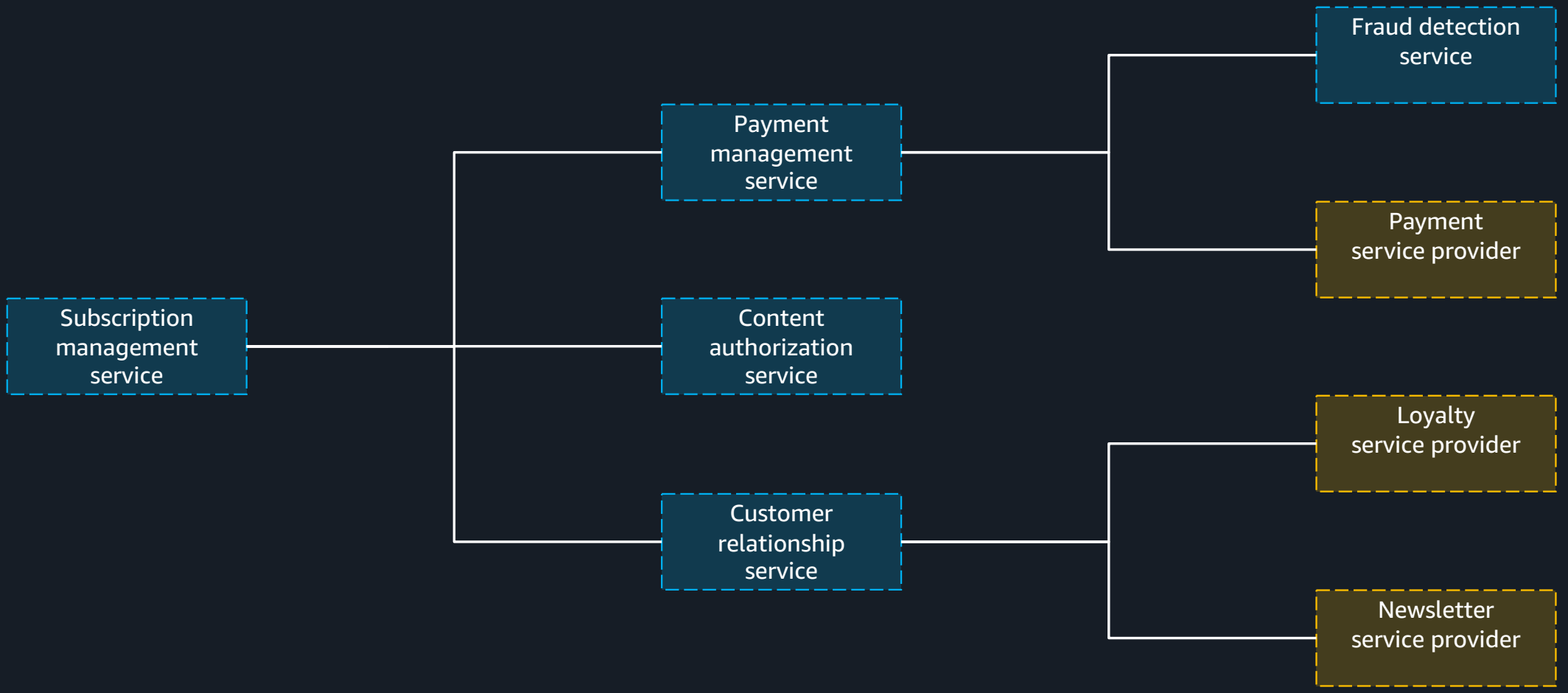
Use case: Subscribe to premium tier



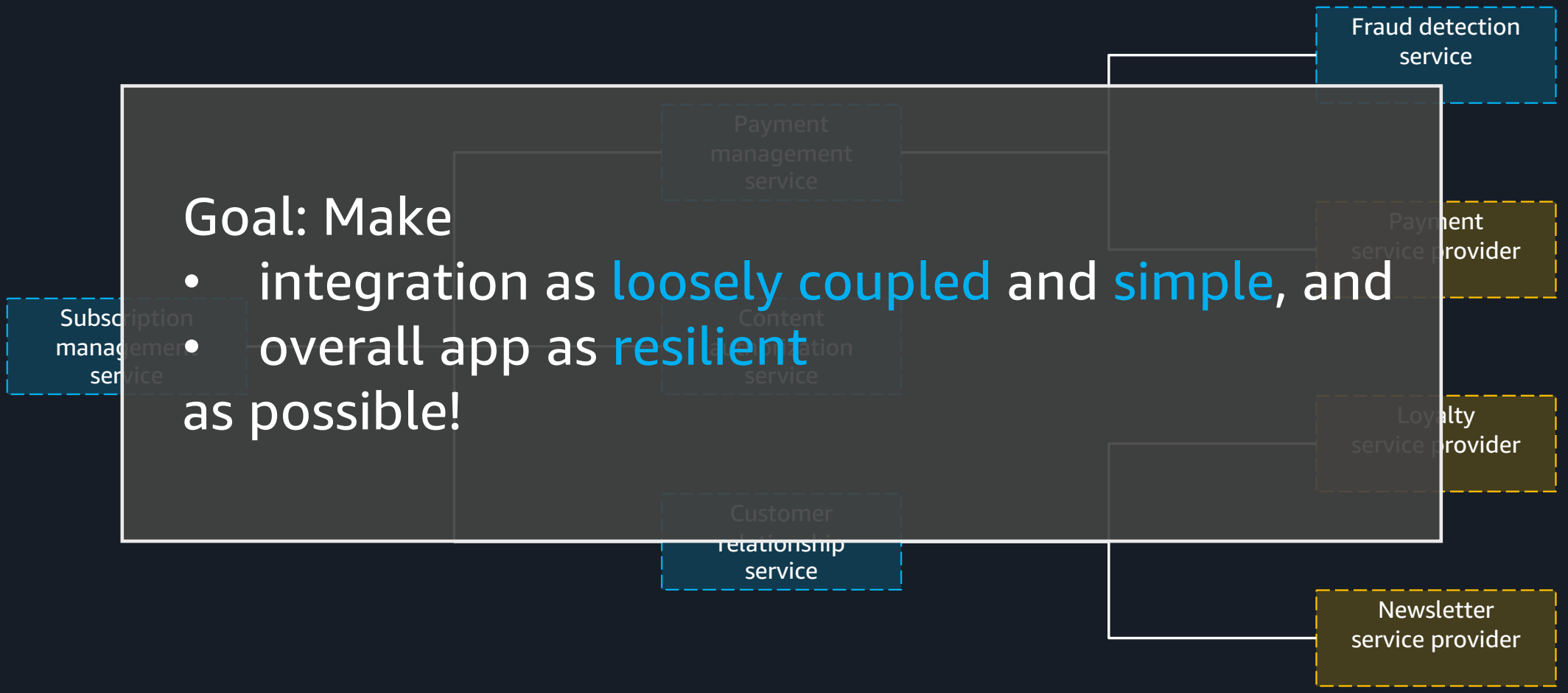
Integrating the downstream systems



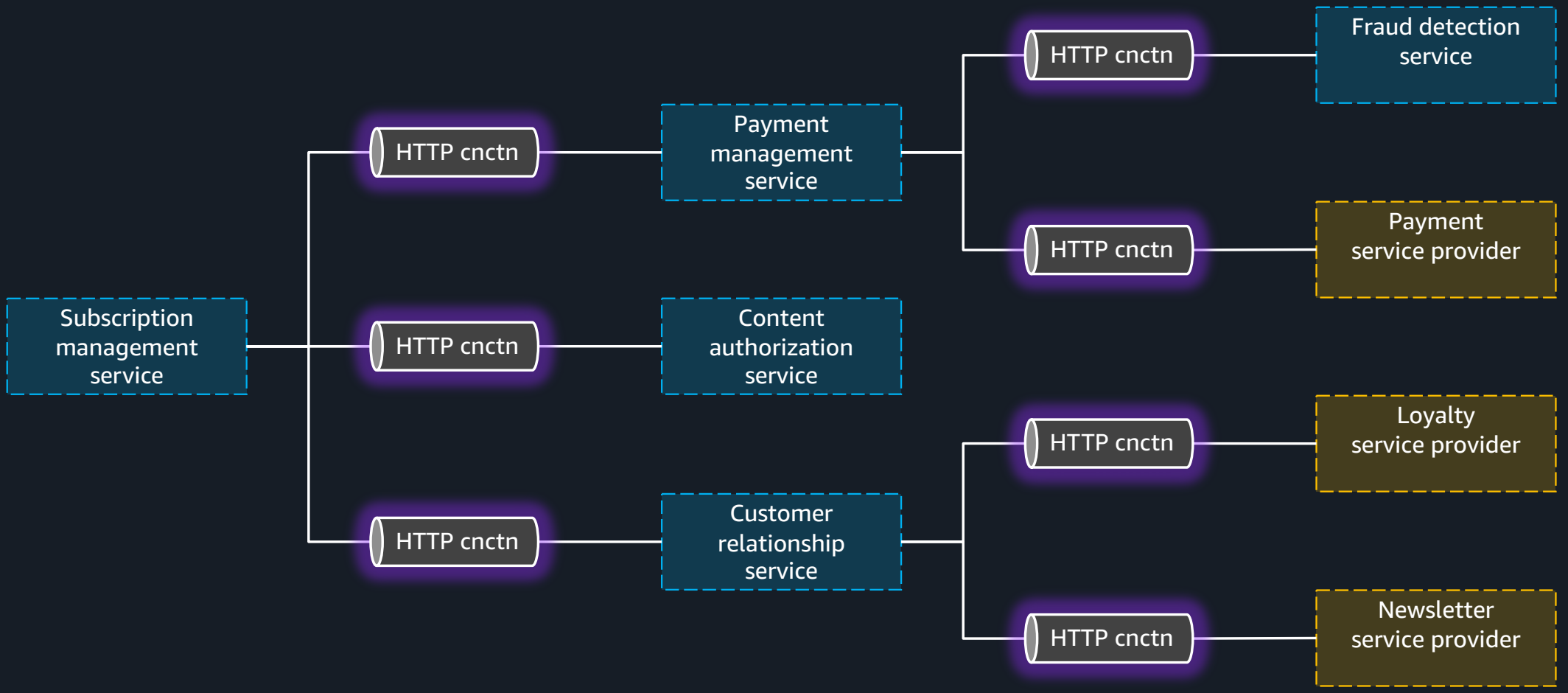
Integrating the downstream systems



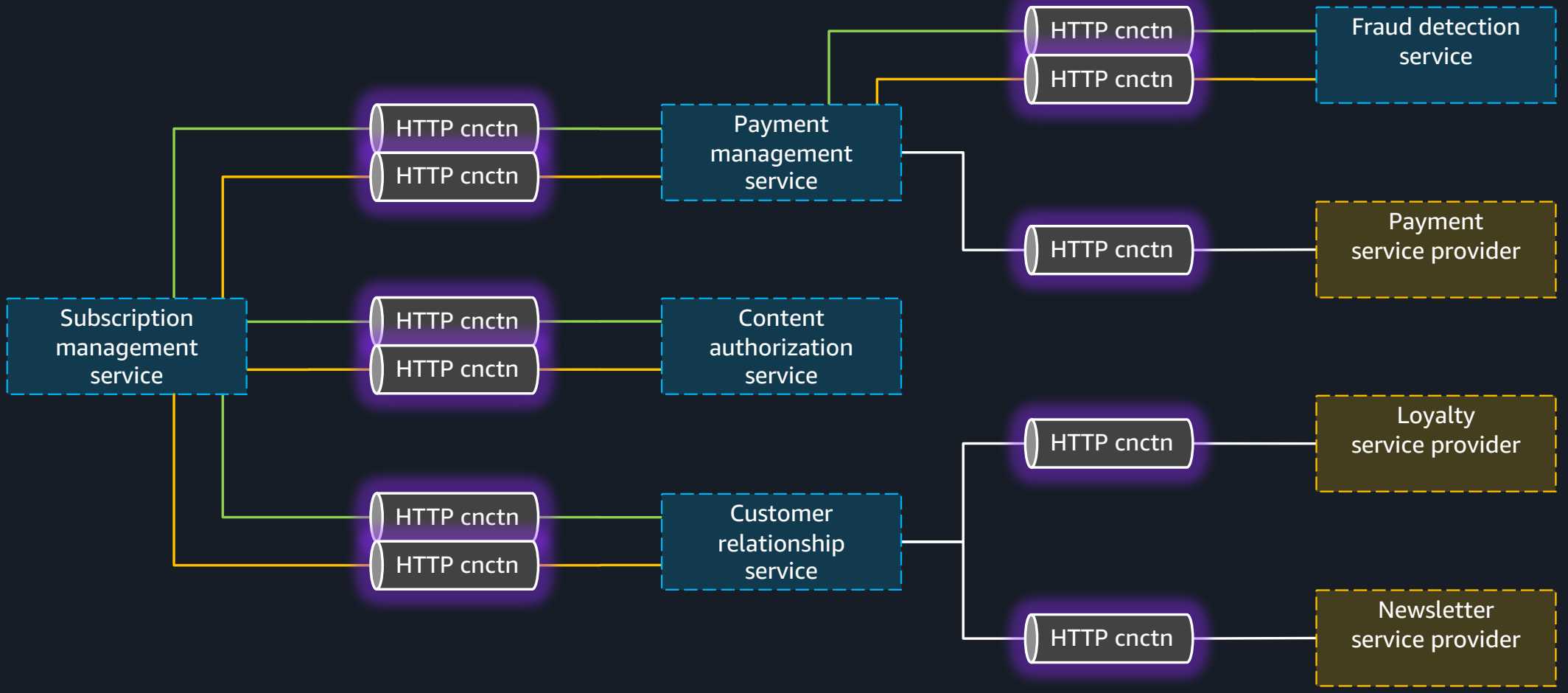
Integrating the downstream systems



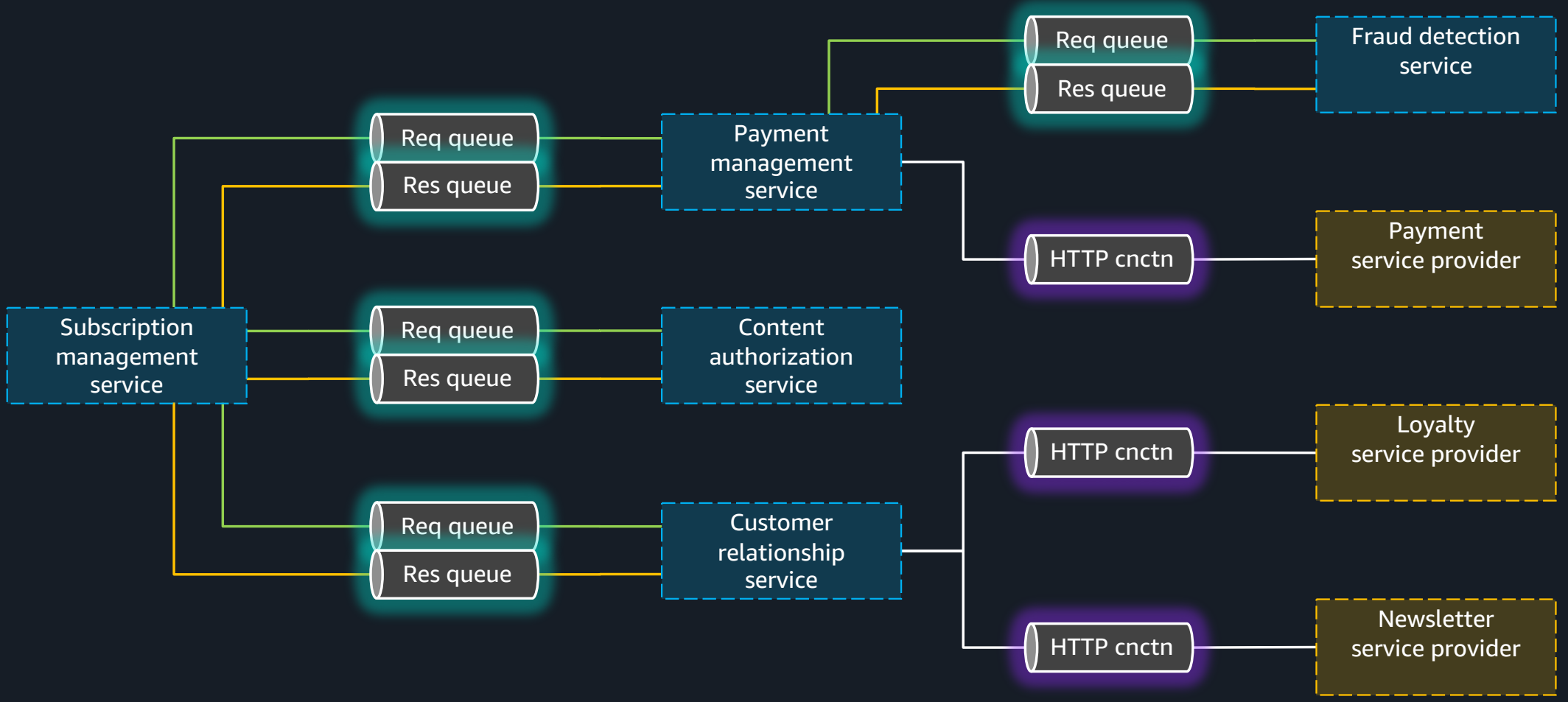
Synchronous request-response w/ APIs



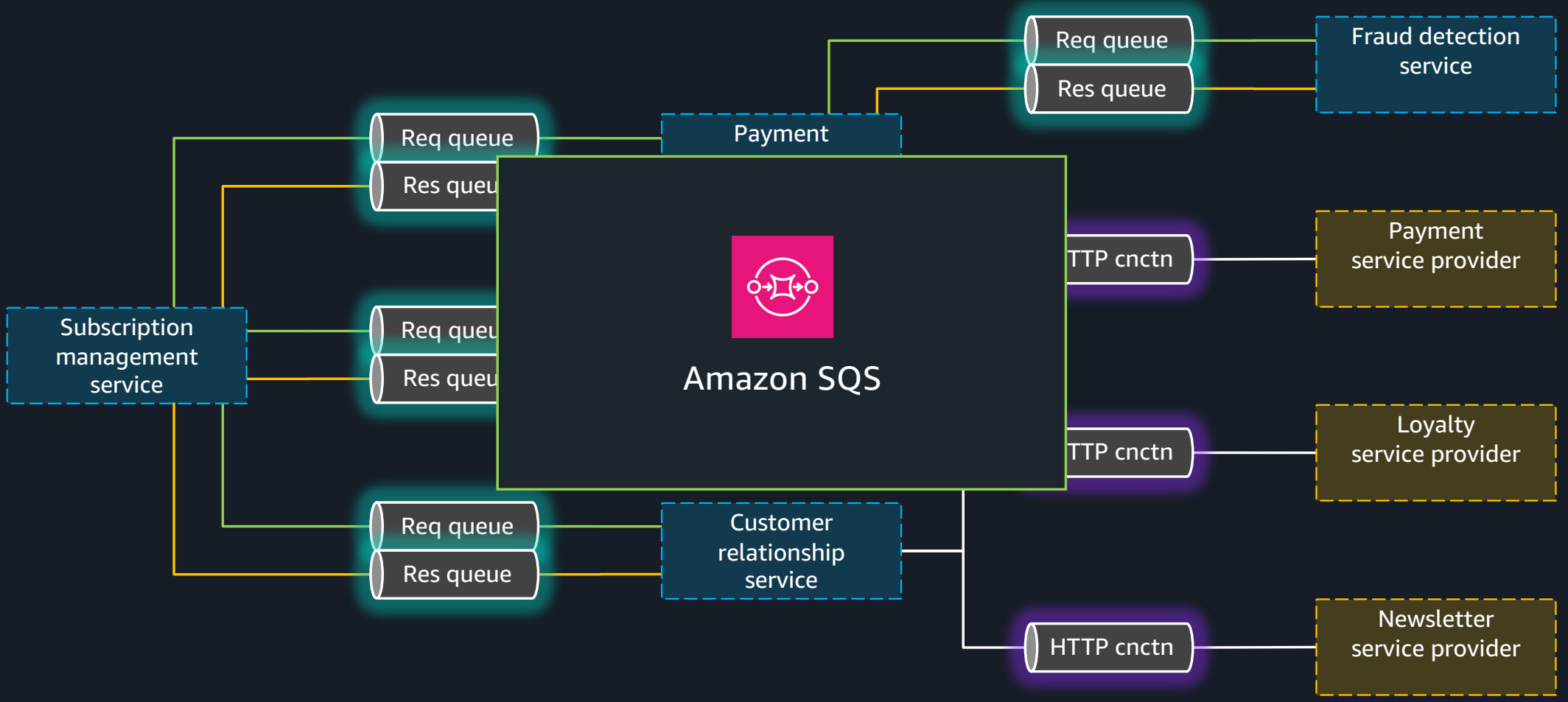
Asynchronous request-response w/ APIs



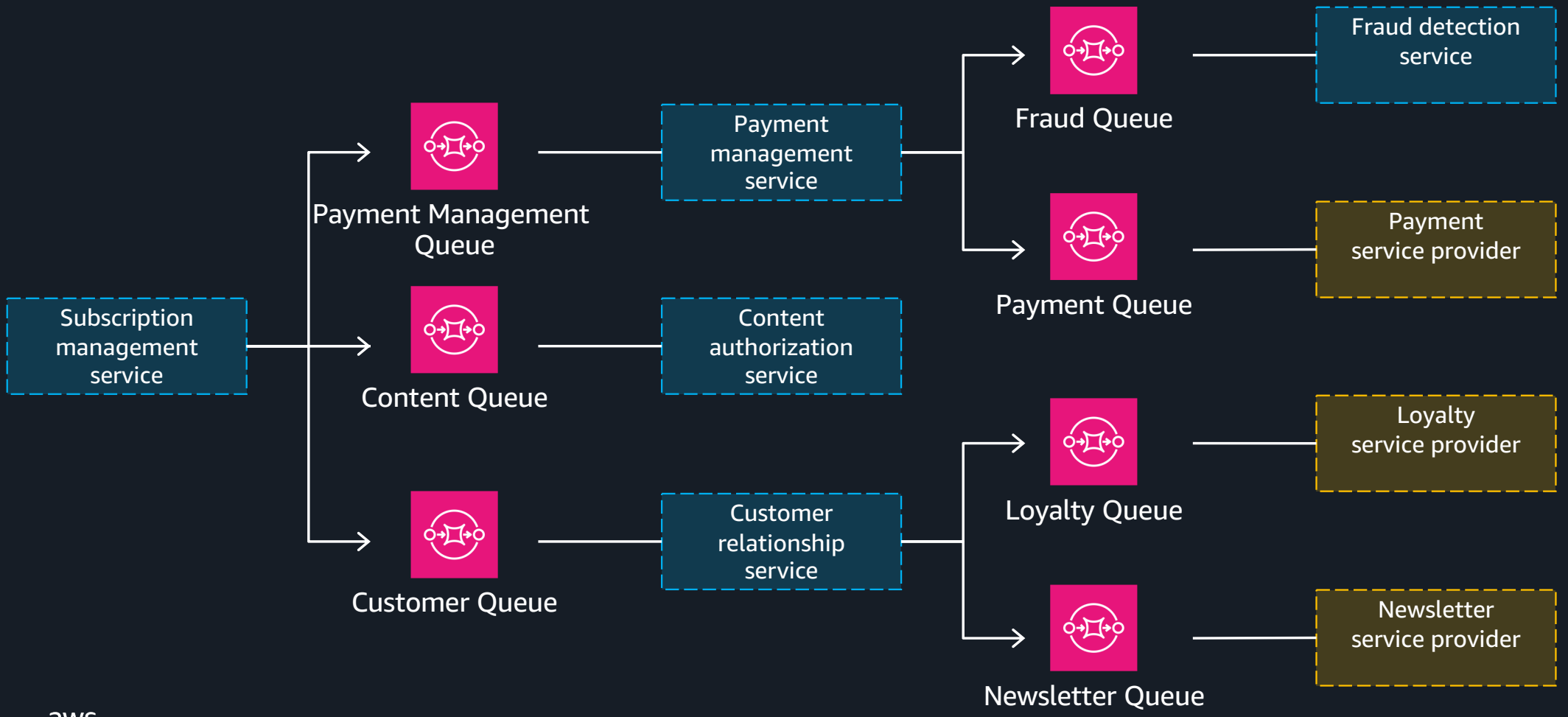
Asynchronous request-response w/ queues



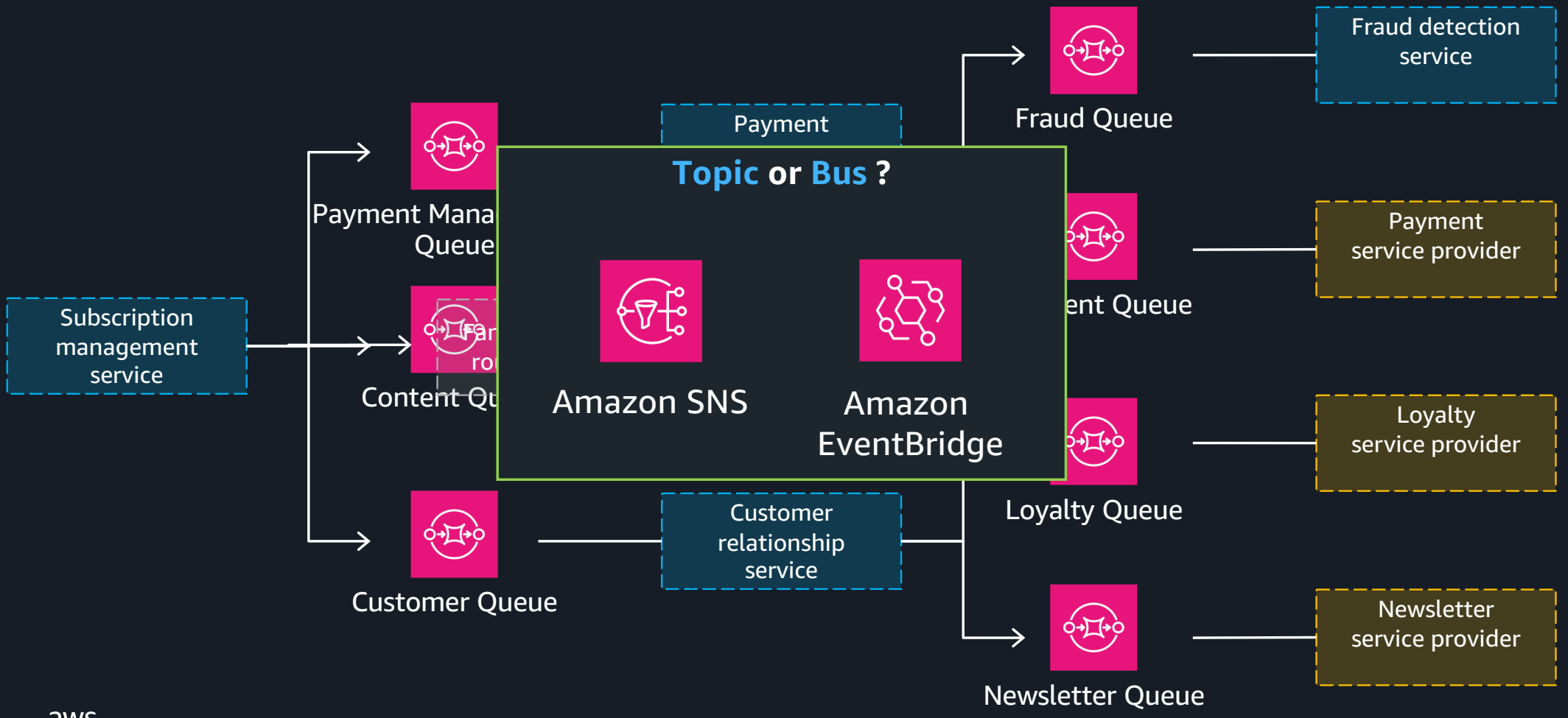
Asynchronous request-response w/ queues



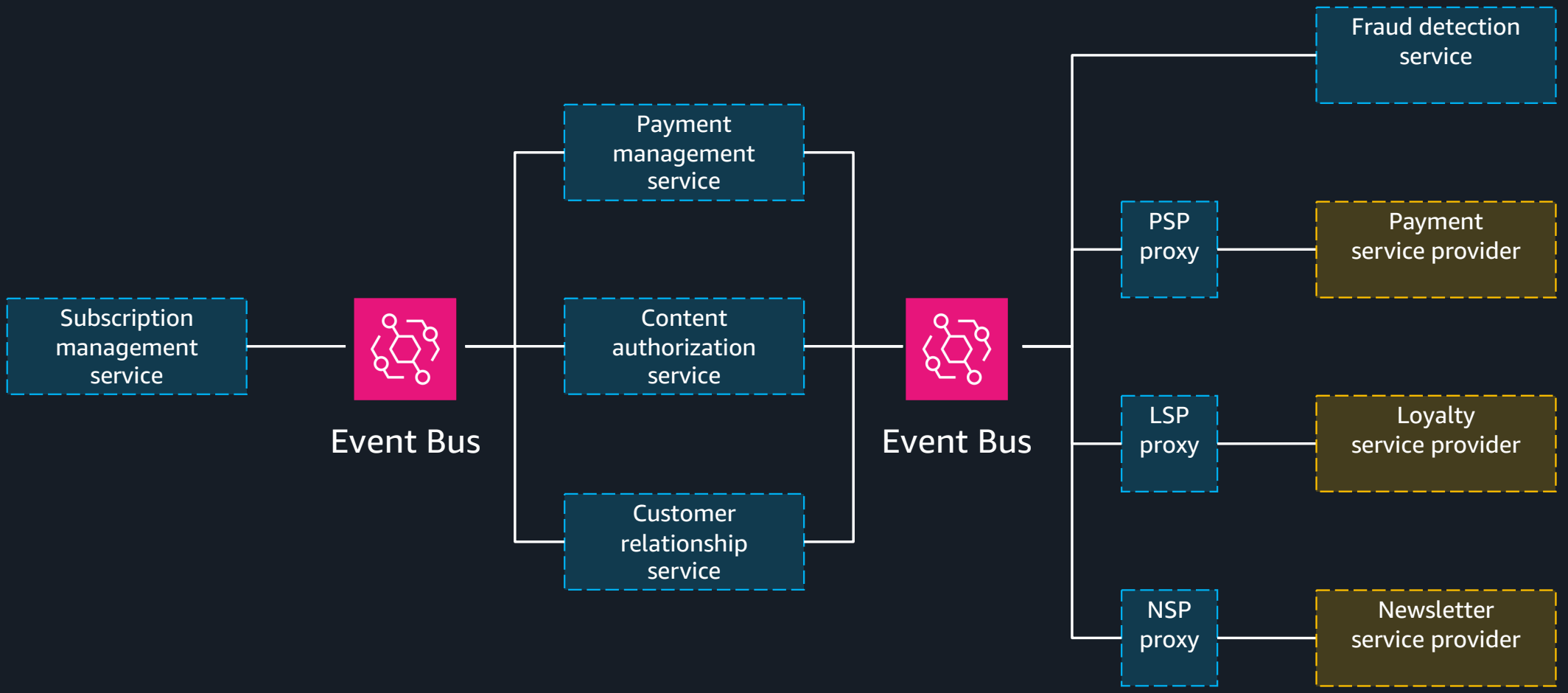
Asynchronous requests with queues



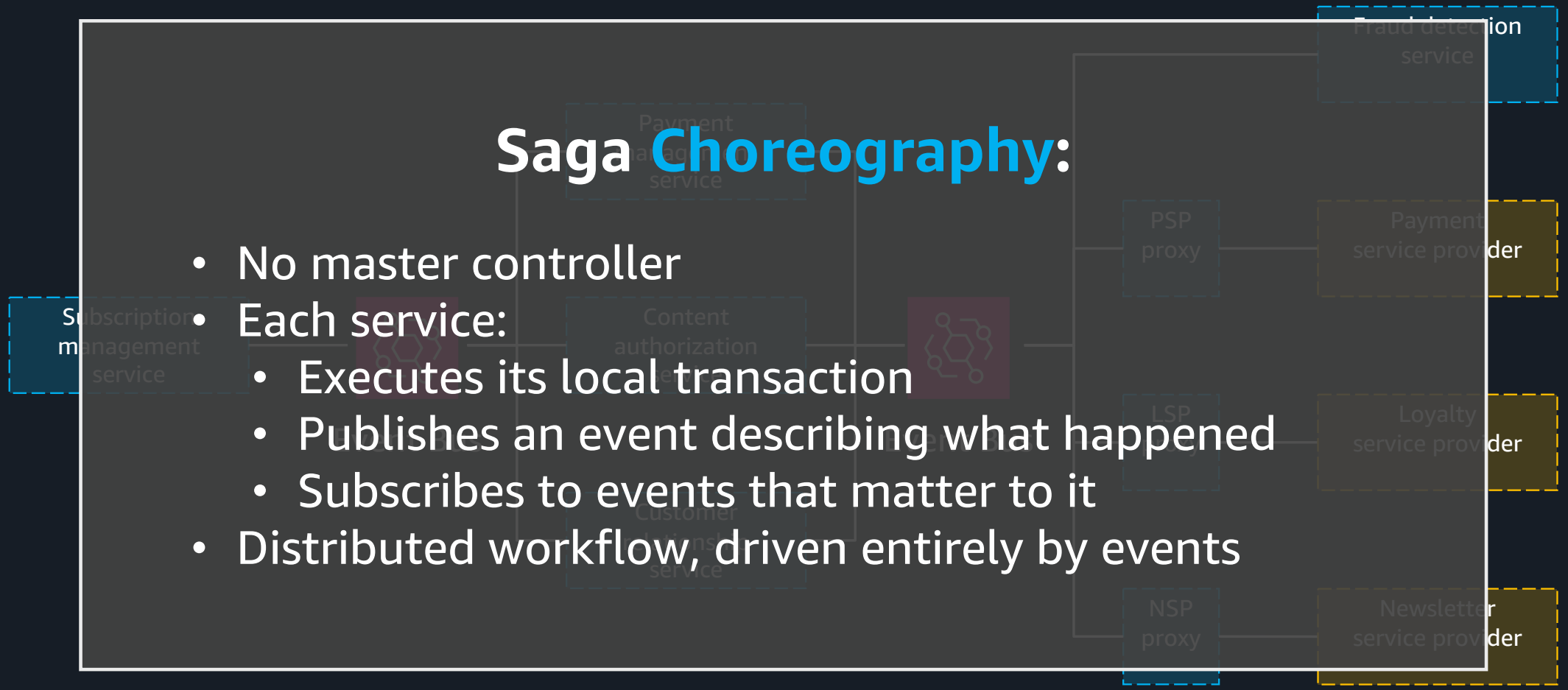
Fan-out chaining with queues



Event-driven interaction with EventBridge



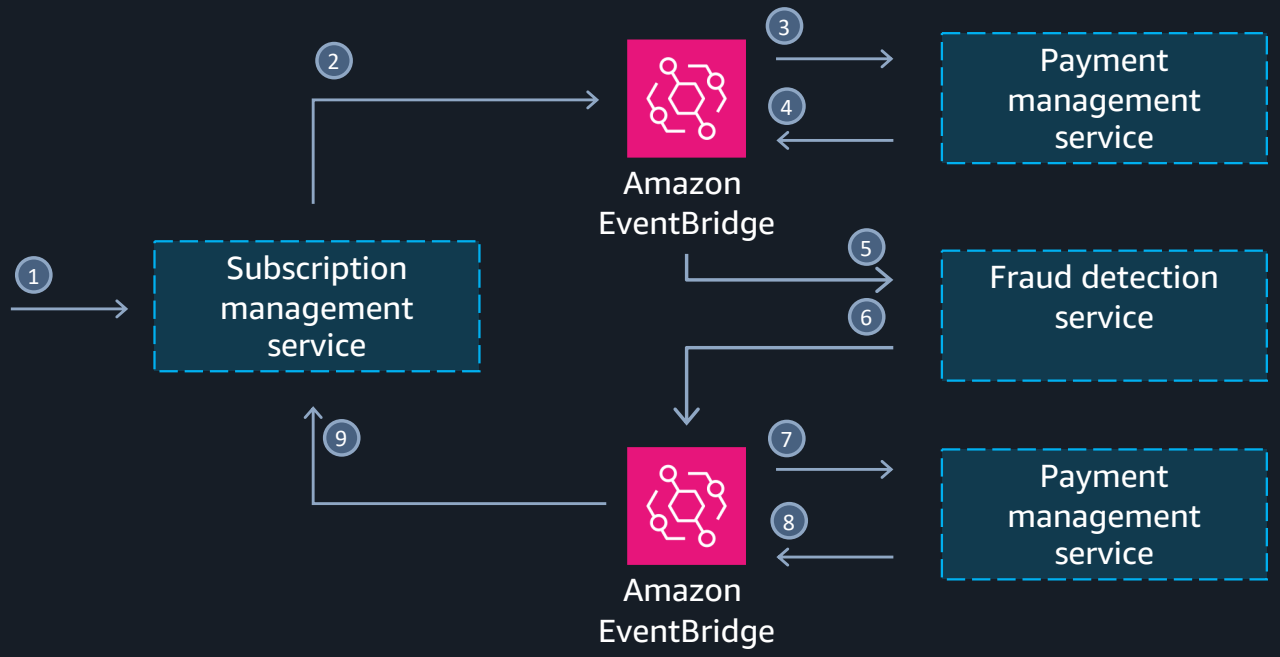
Event-driven interaction with EventBridge



Saga Choreography:

- No master controller
- Each service:
  - Executes its local transaction
  - Publishes an event describing what happened
  - Subscribes to events that matter to it
- Distributed workflow, driven entirely by events

Choreography w/ message bus



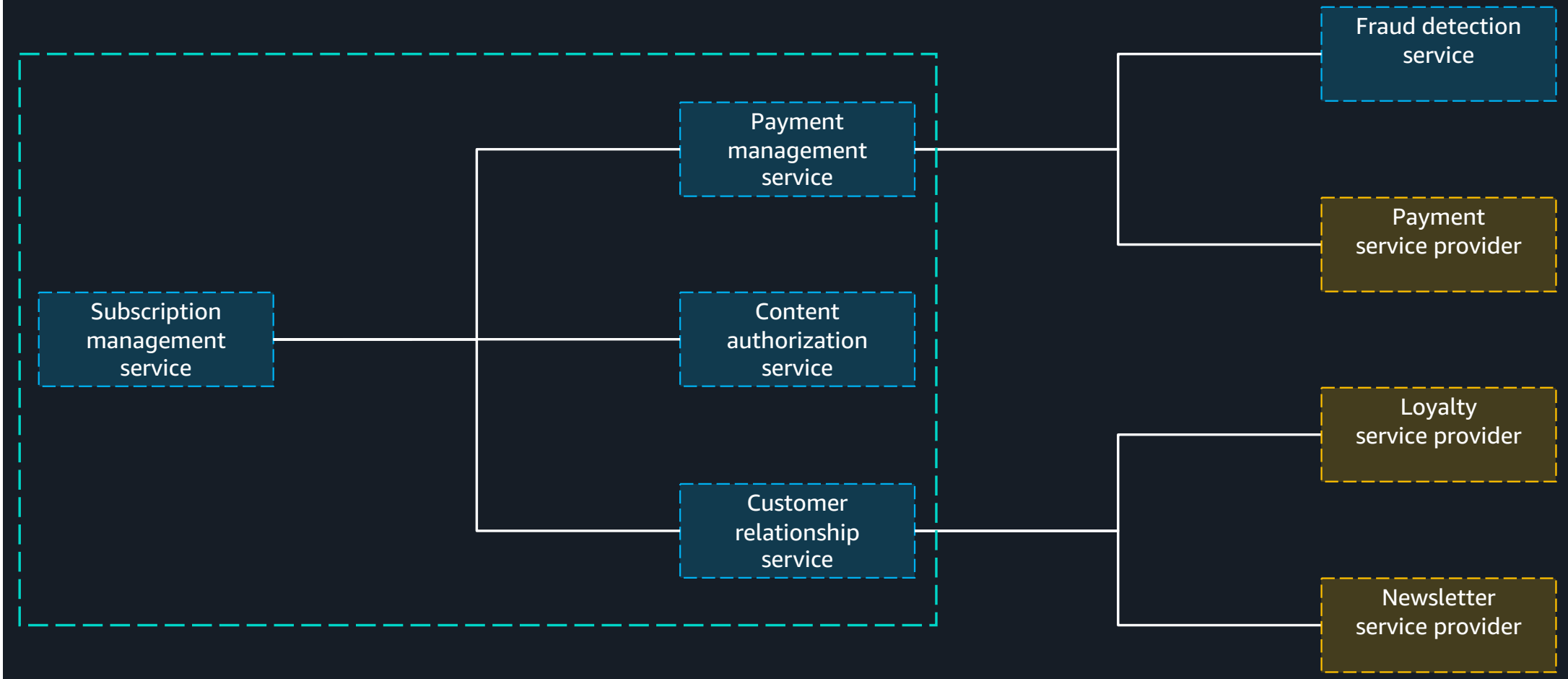
Choreography w/ message bus

Saga Choreography benefits and challenges:

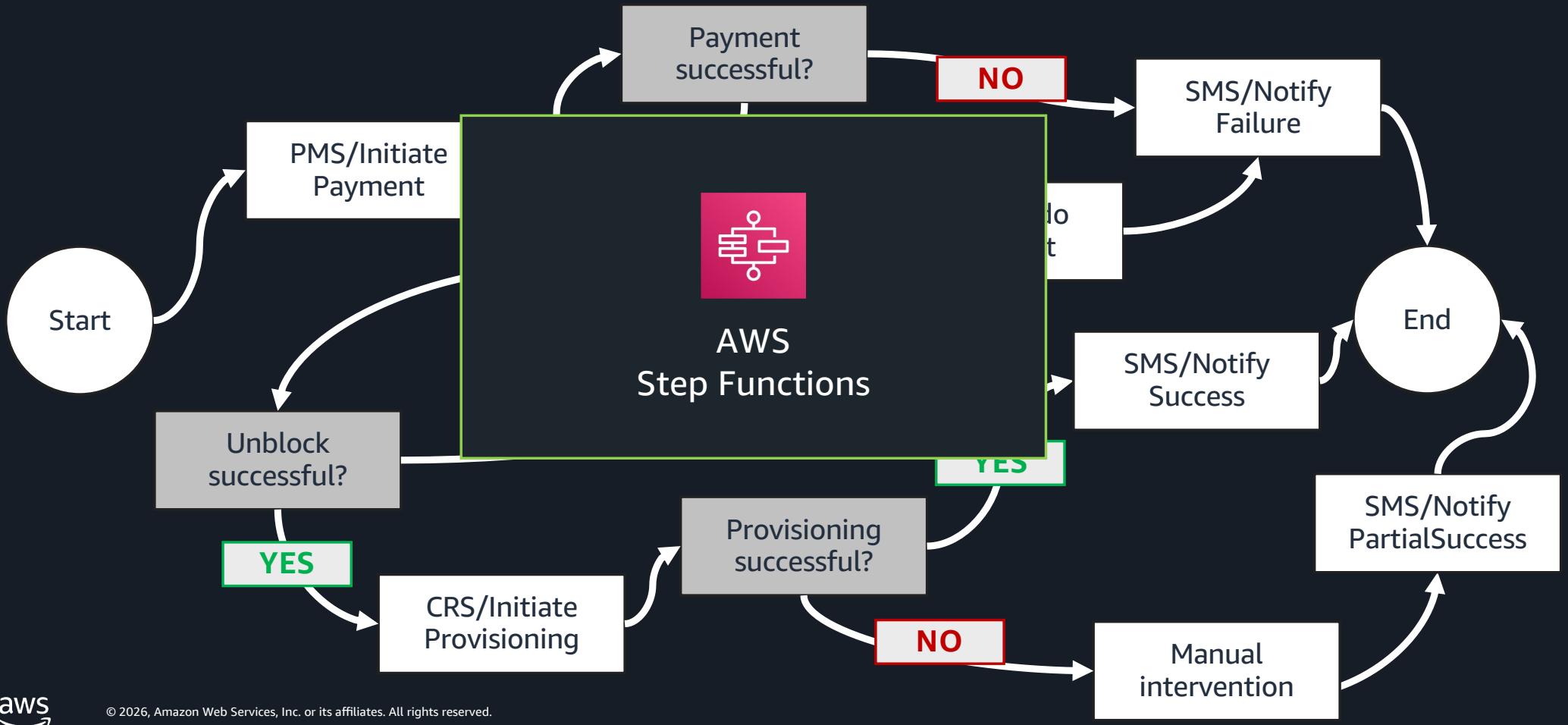
- Easy to get started with (comes up quite naturally)
- Highly scalable (through asynchronous communication)
- Highly decoupled architecture implicitly supports autonomous teams (DDD)
- Simplifies monolith-to-microservices migrations (as we saw)

- Event flow not explicitly visible, but implicitly encoded in systems
- Consequently complex for transactions, compensations / re-driving after failure
- Hard to introduce a new step in the flow – need to touch code and config
- No explicit support for delayed executions and conditions
- Schema changes in flowing events potentially affect all systems

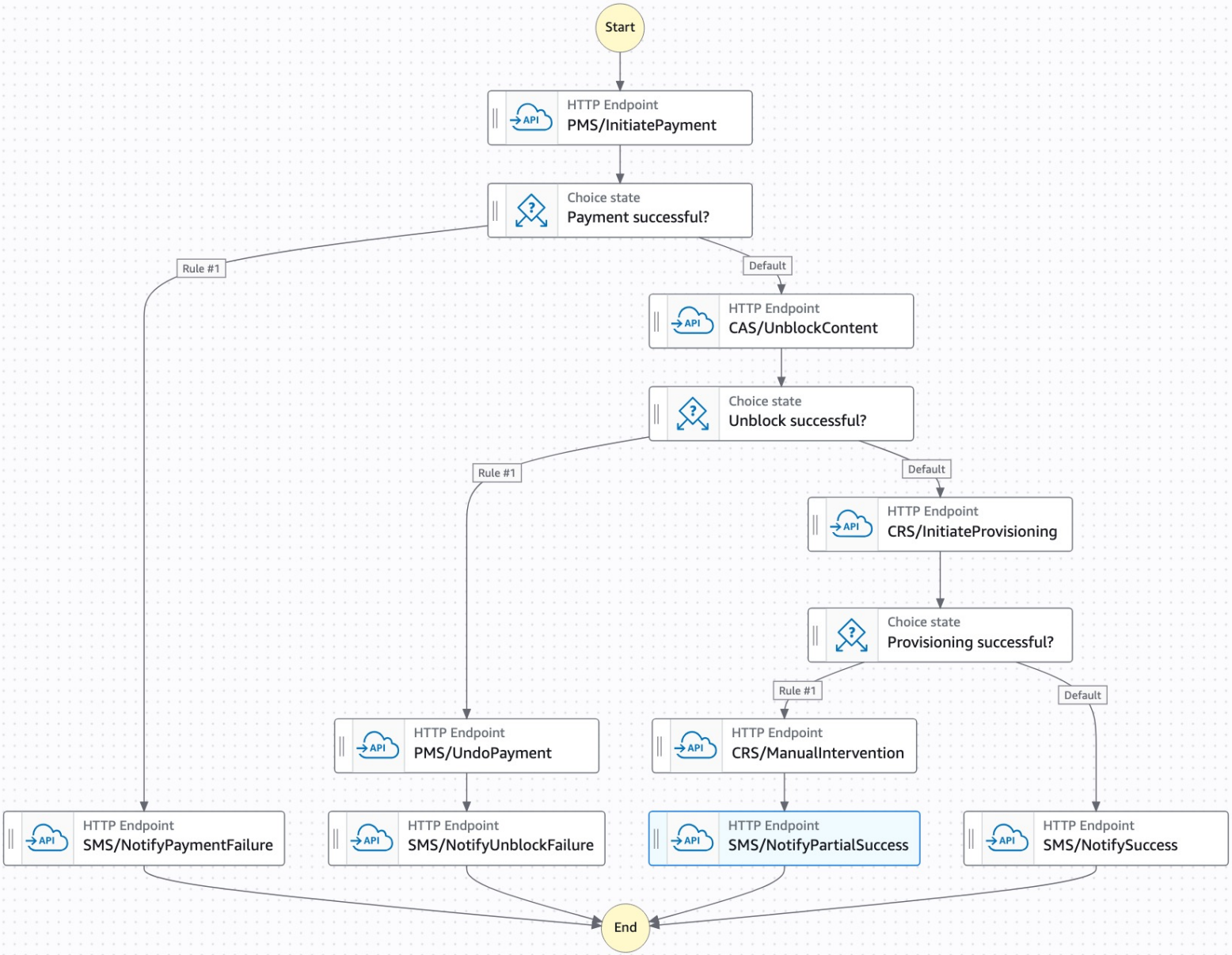
Externalize workflow in an orchestrator



Externalize workflow in an orchestrator



Orchestration with

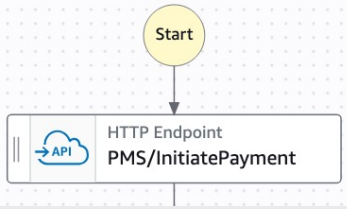


AWS Step Functions

aws

Orchestration with

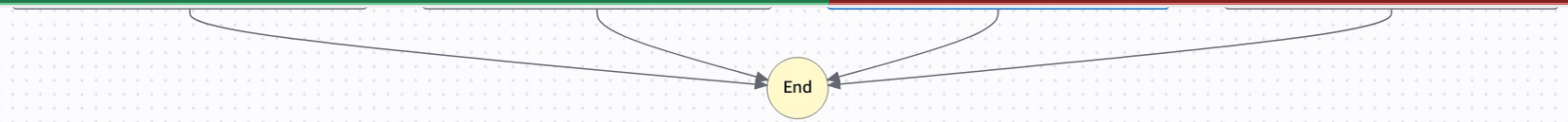
AWS Step Functions



Saga Orchestration benefits and challenges:

- Explicit workflow visibility in orchestrator component
- Simplified debugging due to explicit state during execution
- Easy workflow evolution – no need to touch multiple services
- Better control over consistency due to explicit error handling

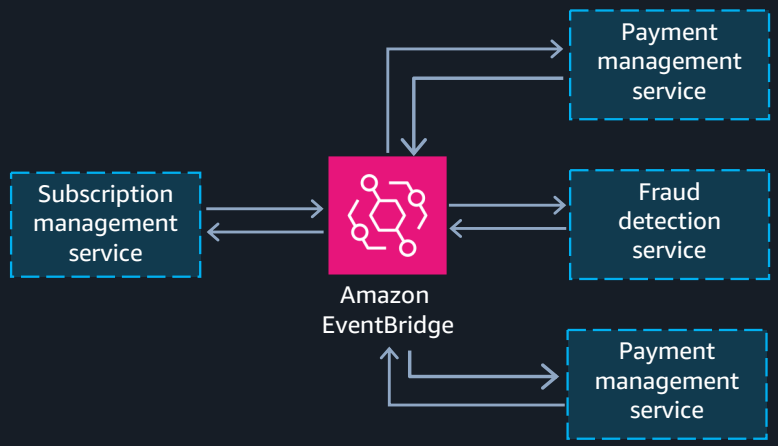
- Introduction of a central dependency that must be HA and HS
- Reduced autonomy: Services become workflow actors, not even-driven agents
- Risk of becoming a god service with poor governance (e.g., leak business rules)
- Tighter coupling at workflow level: Must know participants and their contracts



aws

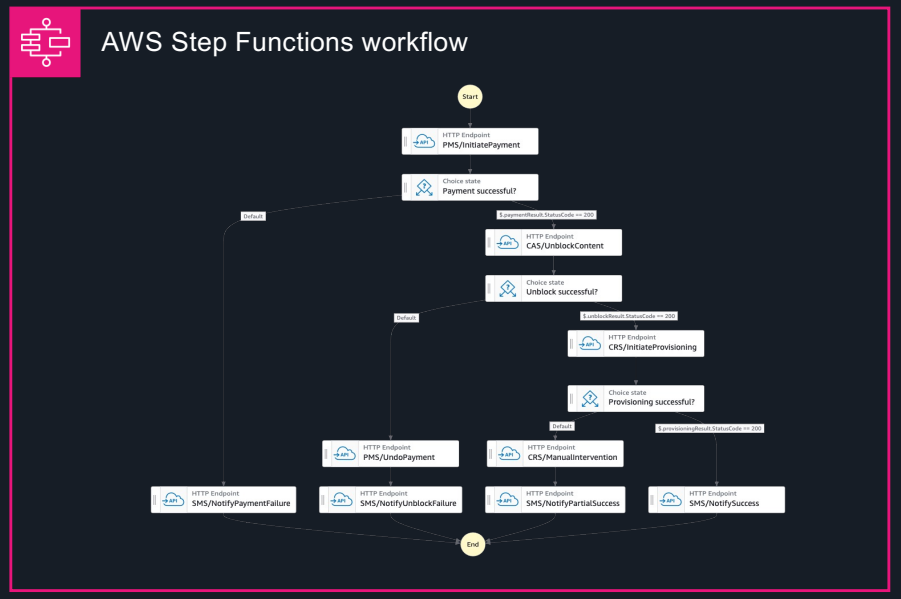
When to use what

Choreography



- ✓ Business logic can evolve easily (add, delete services)
- ✓ Services can scale independently
- ✗ Transactional consistency across services is difficult
- ✗ End-to-end monitoring, handling errors, and timeout are difficult

Orchestration



- ✓ Centralized business logic in orchestrator
- ✓ Transaction management across services
- ✓ End-to-end monitoring and error handling easier
- ✗ Tightly coupled
- ✗ Orchestrator can become single point of failure/limitation
- ✗ Single point of maintenance (can be good or bad)

When to use what

Prefer Choreography when:

- Your system is naturally event-driven
- Workflows are simple
- Teams require high autonomy
- Organizational decentralization is key
- Failure complexity is acceptable

“Let each service mind its own business.”

Prefer Orchestration when:

- Workflows are long-running or complex
- Business correctness matters more than autonomy
- Governance or compliance require visibility
- Need for centralized retries, compensation, rules, and timeouts

“Make process behaviour obvious, testable, and evolvable.”

When to use what

Prefer Choreography when:

- Your sy
- Workflo
- Teams
- Organiz
- Failure

"Let each s

Prefer Orchestration when:

mplex

e than

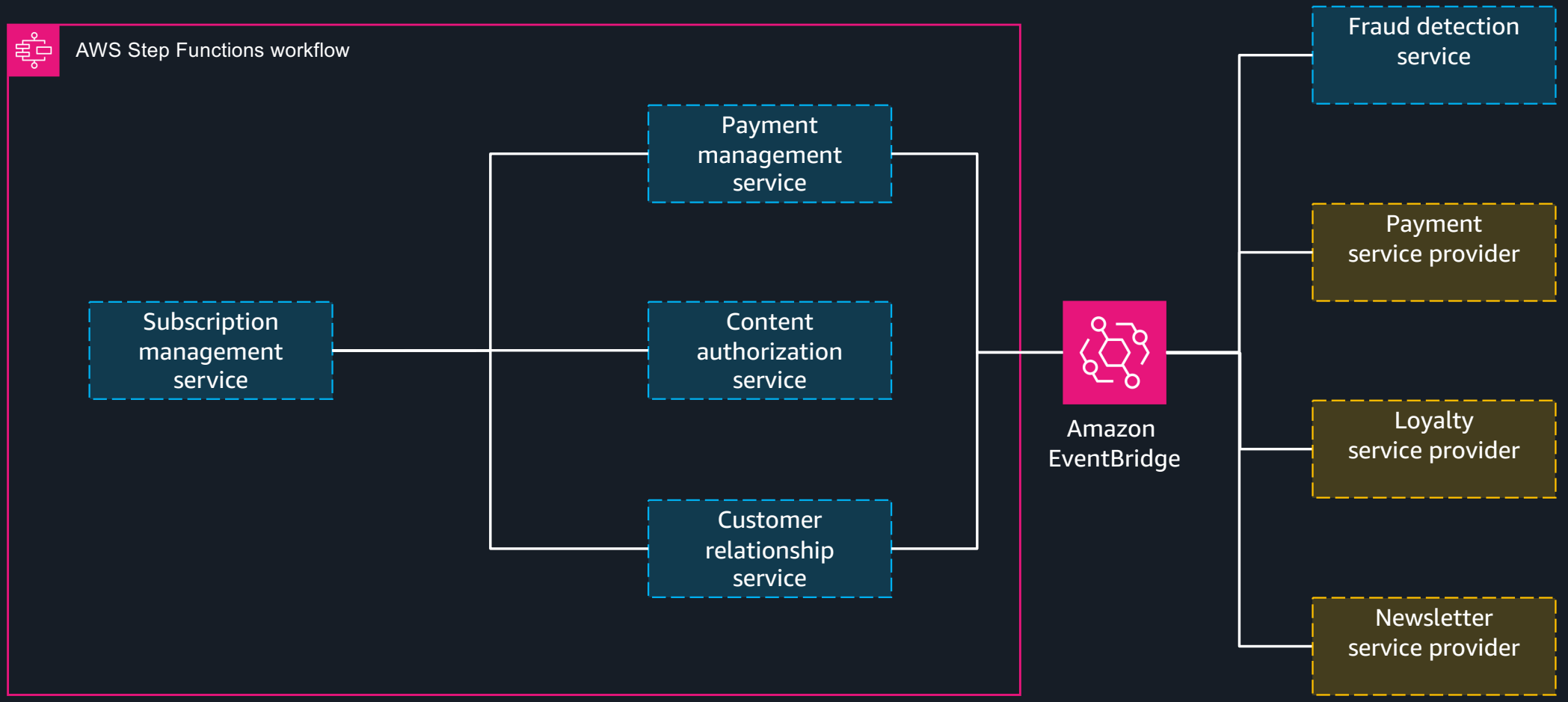
e visibility

ensation,

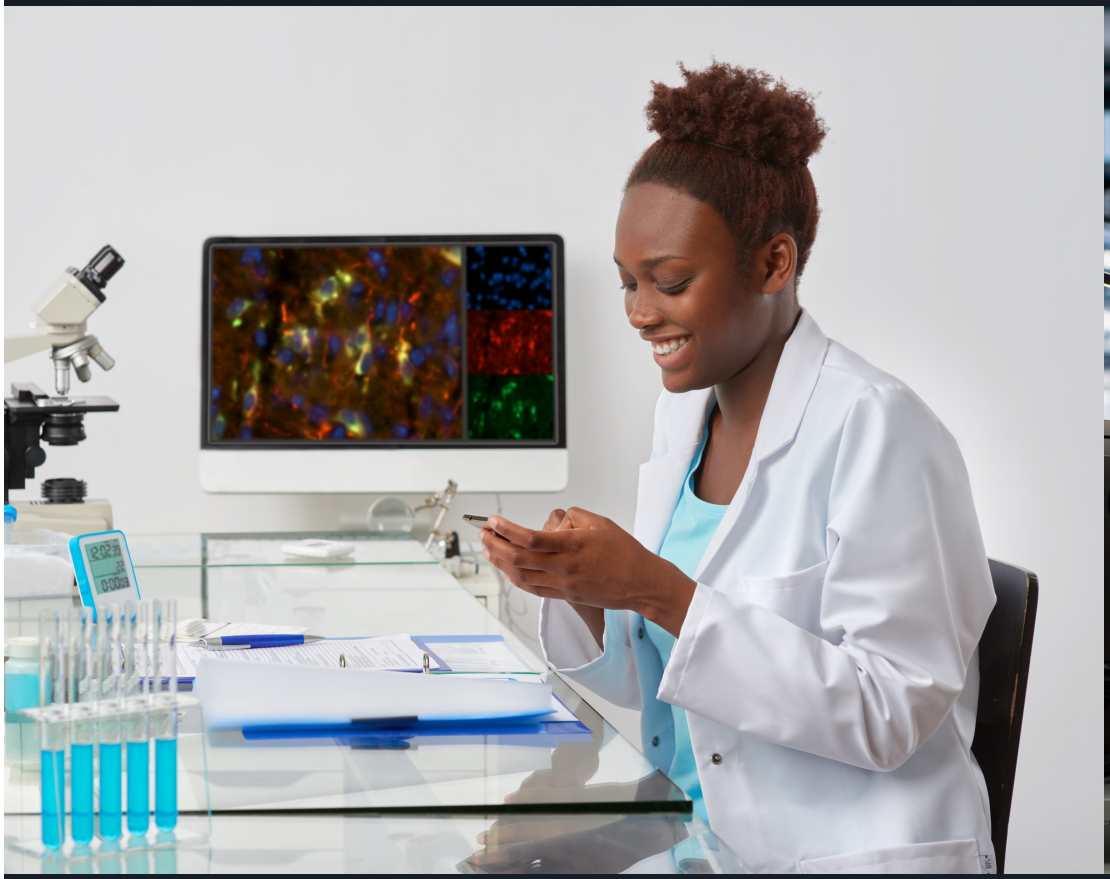
"Make process behaviour obvious, testable, and evolvable."

Combine
Choreography and Orchestration
for the best of both worlds

Externalize workflow in an orchestrator



Suffering from success



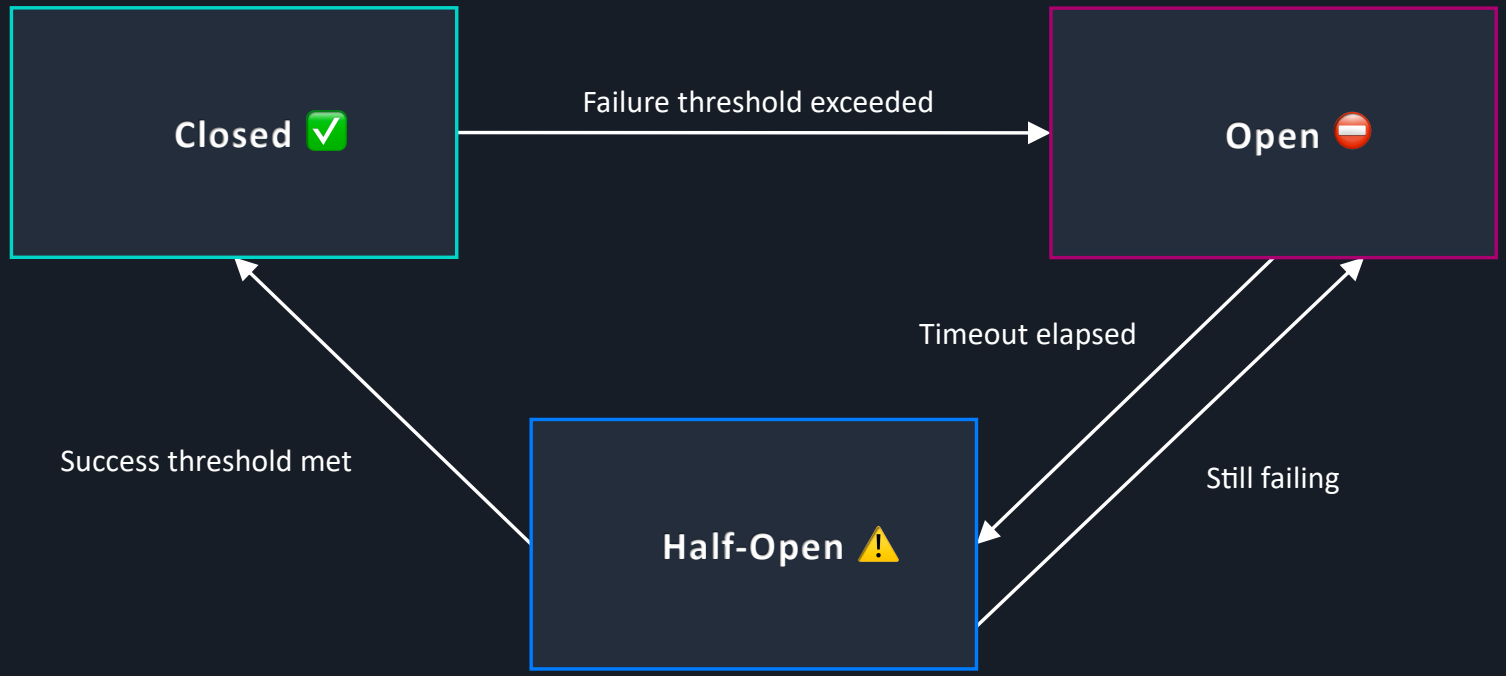
Our customers

The subscriptions team

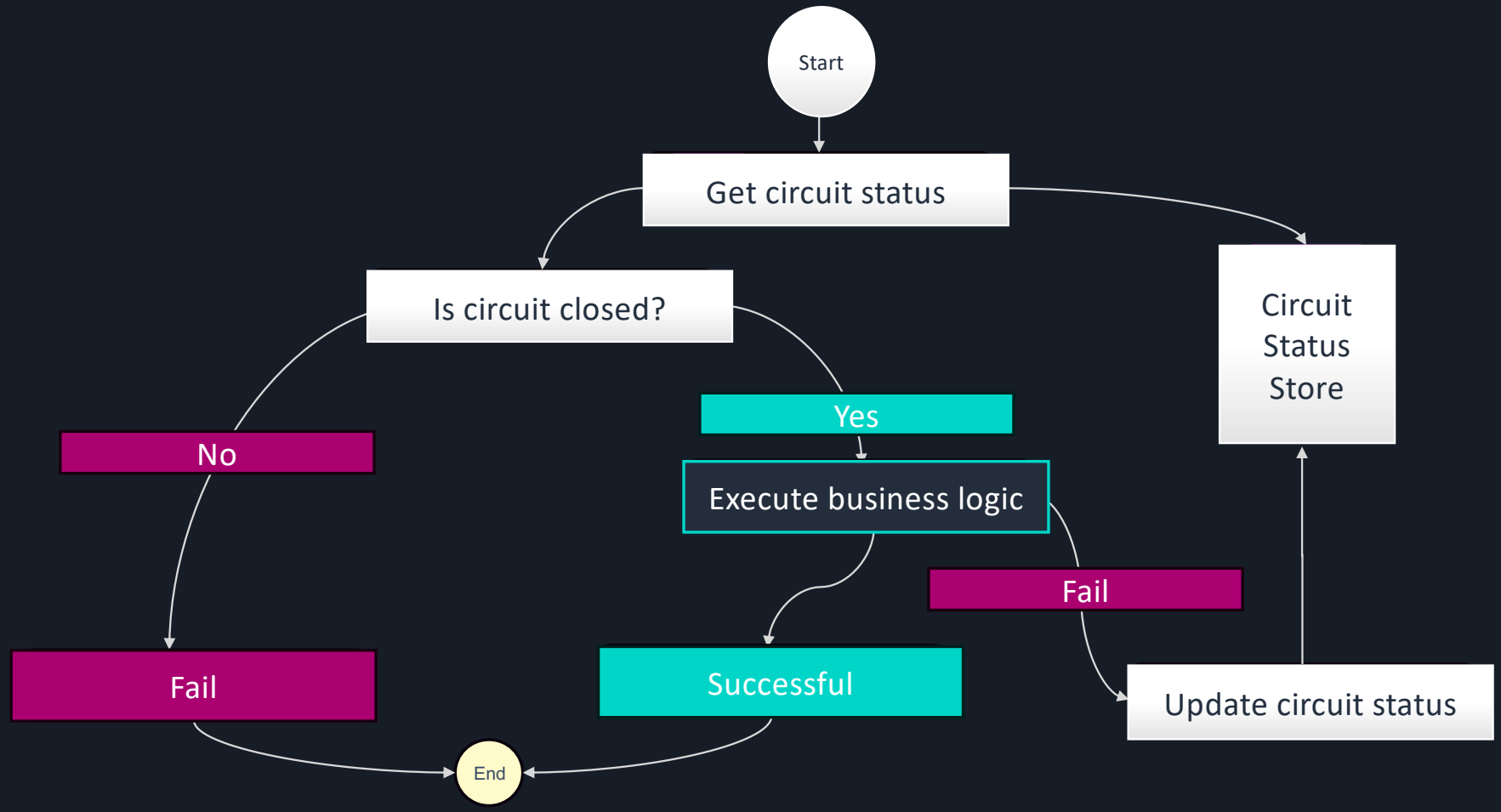
Circuit Breaker pattern

Prevent repeated execution of operations that are likely to fail and protect distributed systems from overload and cascading failure.

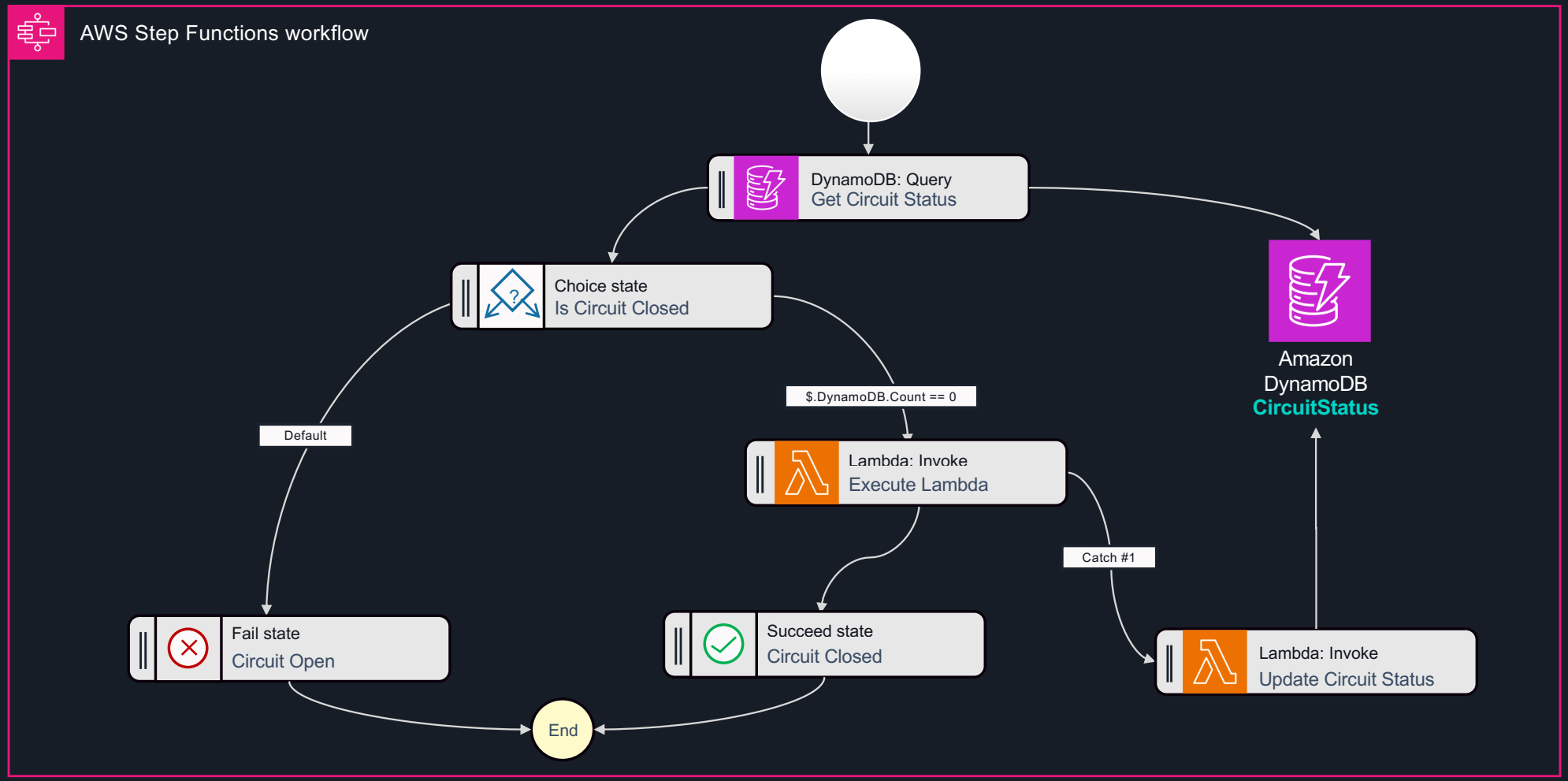
Circuit Breaker state transitions



A Circuit Breaker Protected Execution



A Circuit Breaker Protected Execution



The “Subscription Crisis” meeting

On-Call
”How are we supposed to know when to start the processing again? We can’t spend all time testing this every 5 minutes.”

CFO
“New subscribers are critical in our current growth phase!”

Ops Lead
“The subscription workflow fails at high load with different errors.”

User Experience
”Our subscribers need to be informed about delays or support will be overrun.”

Architect
”How about adding subscribers to a backlog but pausing when the systems are overloaded?”

An event-driven Circuit Breaker approach

On-Call
”How are we supposed to know when to start the processing again? We can’t spend all time testing this every 5 minutes.”

CFO
“New subscribers are critical in our current growth phase!”

AWS Step Functions
Subscription State Machine

User Experience
”Our subscribers need to be informed about delays or support will be overrun.”

Architect
”How about adding subscribers to a backlog but pausing when the systems are overloaded?”

Amazon CloudWatch -
Subscriptions Failing Alarm

aws

An event-driven Circuit Breaker approach

On-Call
”How are we supposed to know when to start the processing again? We can’t spend all time testing this every 5 minutes.”

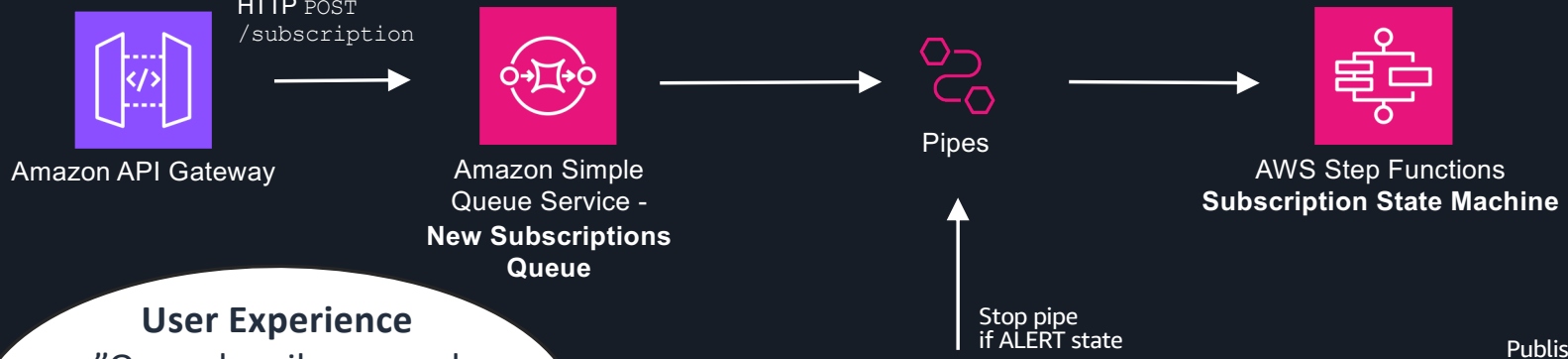
User Experience
”Our subscribers need to be informed about delays or support will be overrun.”

Architect
”How about adding subscribers to a backlog but pausing when the systems are overloaded?”

Amazon CloudWatch - **Subscriptions Failing Alarm**

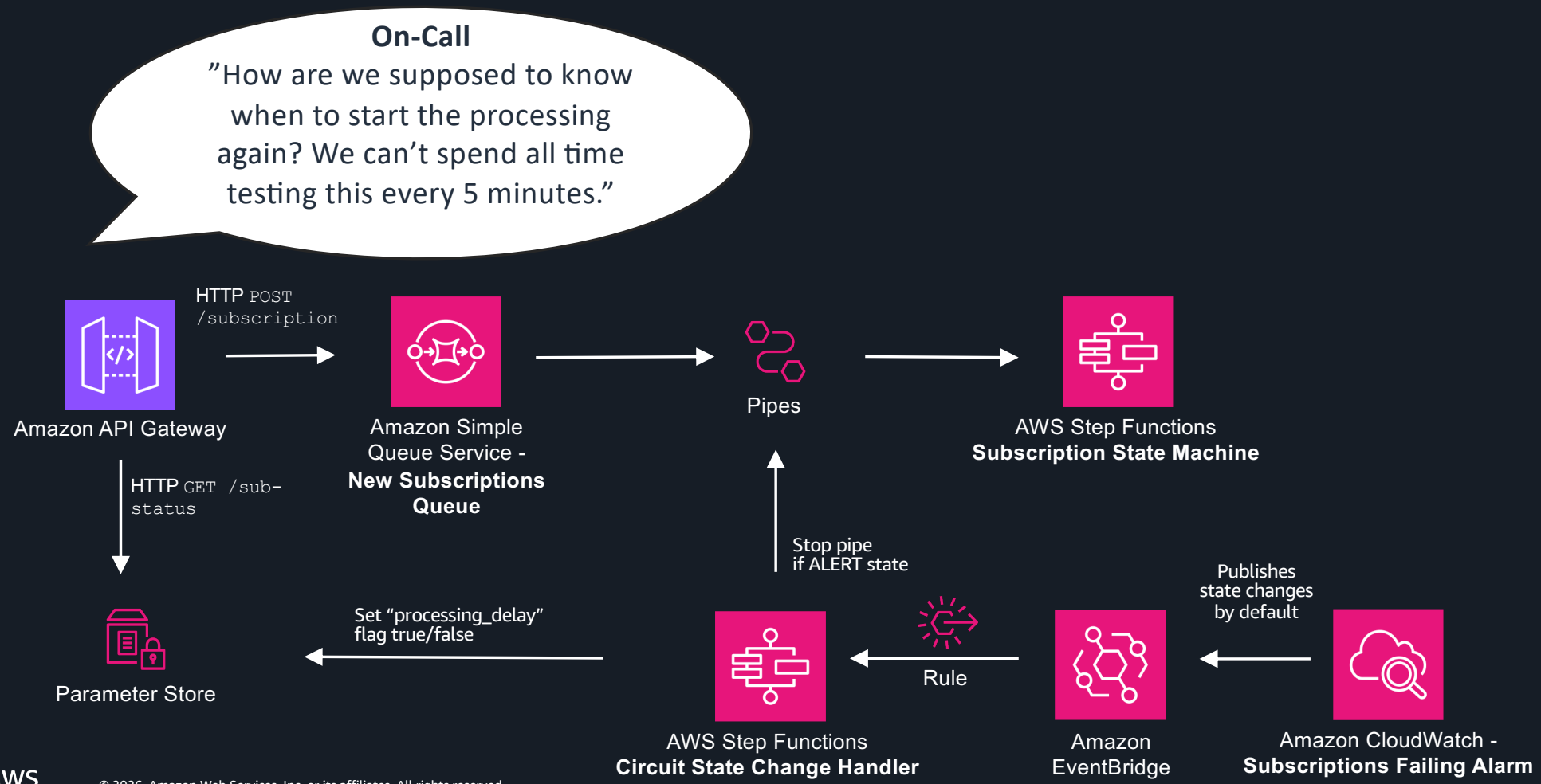
An event-driven Circuit Breaker approach

On-Call
”How are we supposed to know when to start the processing again? We can’t spend all time testing this every 5 minutes.”

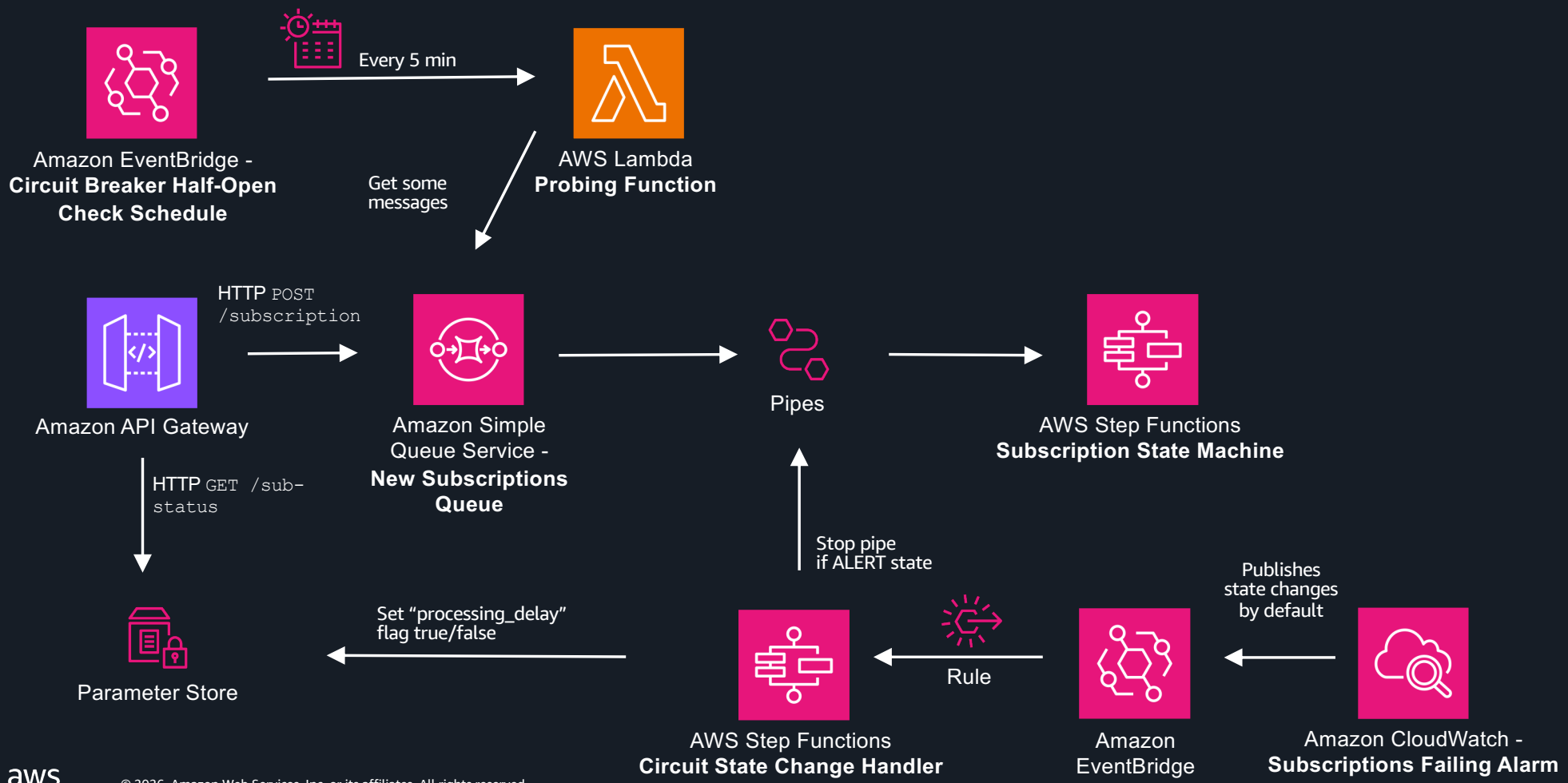


User Experience
”Our subscribers need to be informed about delays or support will be overrun.”

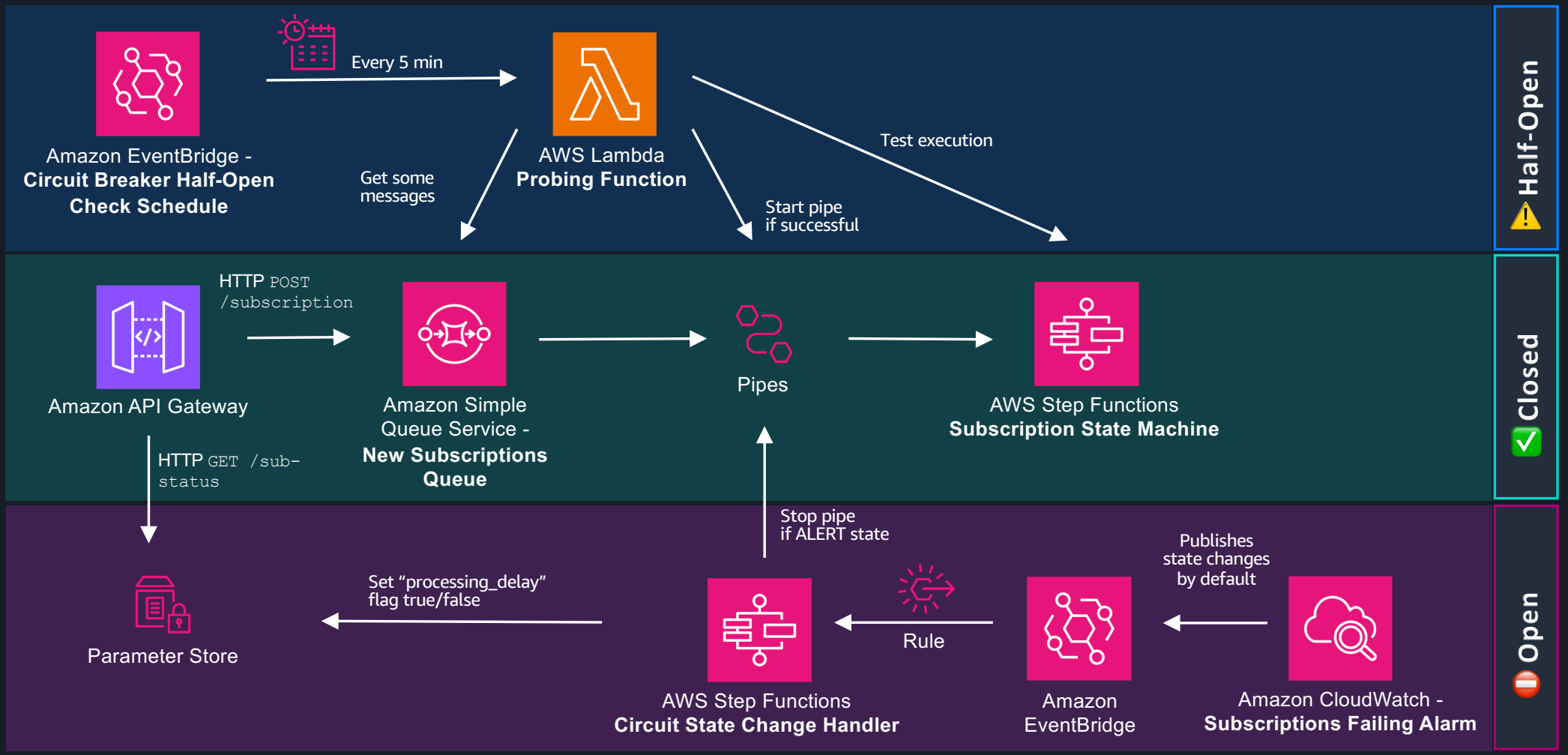
An event-driven Circuit Breaker approach



An event-driven Circuit Breaker approach



An event-driven Circuit Breaker approach



Actions and resources

Dive Deeper into Modernization Patterns

AWS Prescriptive Guidance

Cloud design patterns, architectures, and implementations

http://go.aws/49qBn9Y

AWS Blog Post

Using the circuit-breaker pattern with AWS Lambda extensions and Amazon DynamoDB

https://go.aws/3P5PZEc

Speaker Contact & Presentation Resources

AWS Workshop

Building near real-time LLM-powered features for monolithic applications

https://catalog.workshops.aws/build-real-time-features-legacy-apps