
AWS
re:Invent
NOV. 28 – DEC. 2, 2022 | LAS VEGAS, NV

ARC207

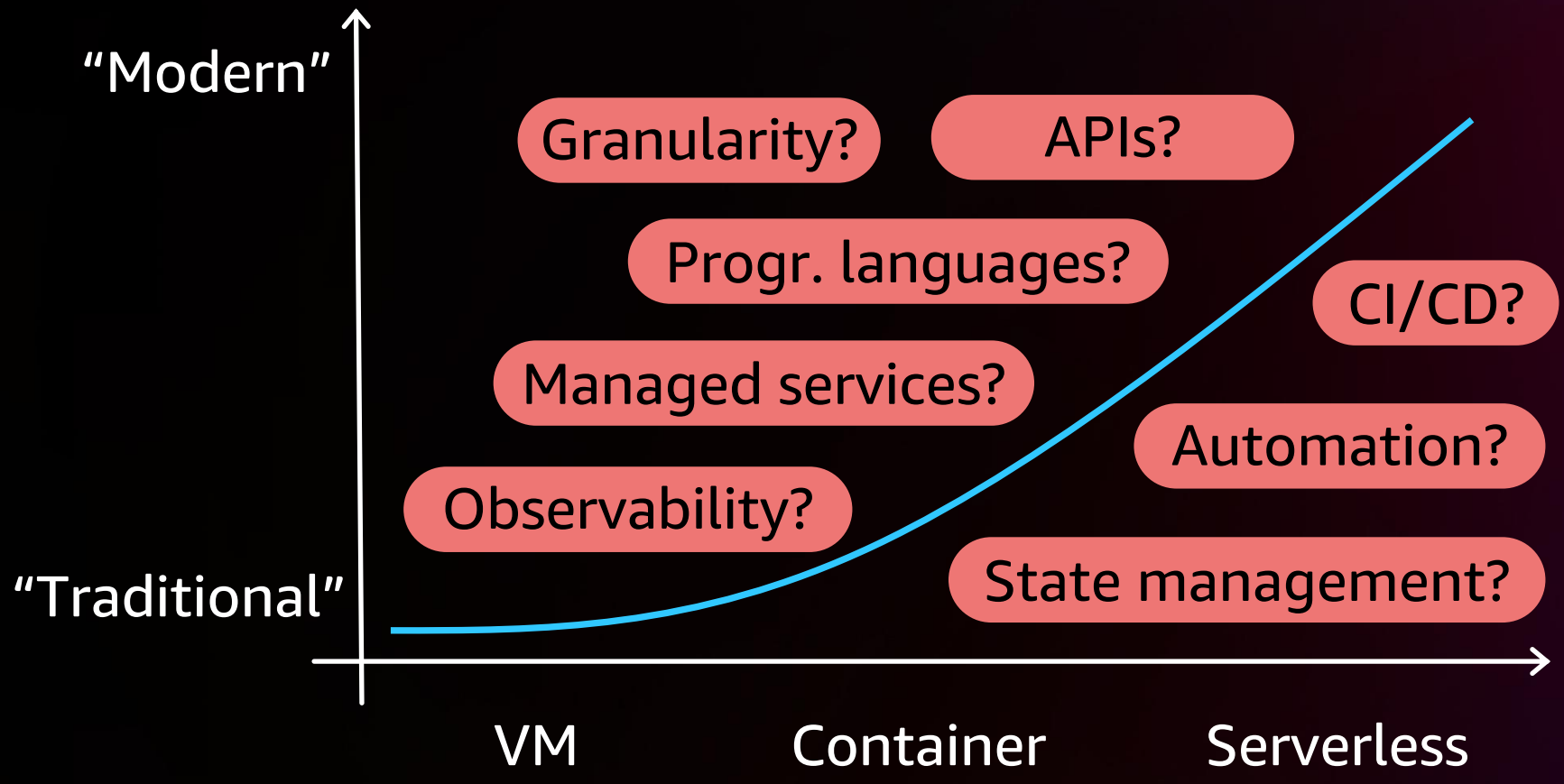
Modern cloud applications: Do they lock you in?

Gregor Hohpe

Enterprise Strategist
Amazon Web Services

Modern cloud applications

Modern isn’t just the runtime



Serverless – Not just a runtime

Compute


AWS Lambda


AWS Fargate

Data stores


Amazon S3


Amazon Aurora Serverless


Amazon DynamoDB

Integration


Amazon EventBridge


Amazon API Gateway


Amazon SQS

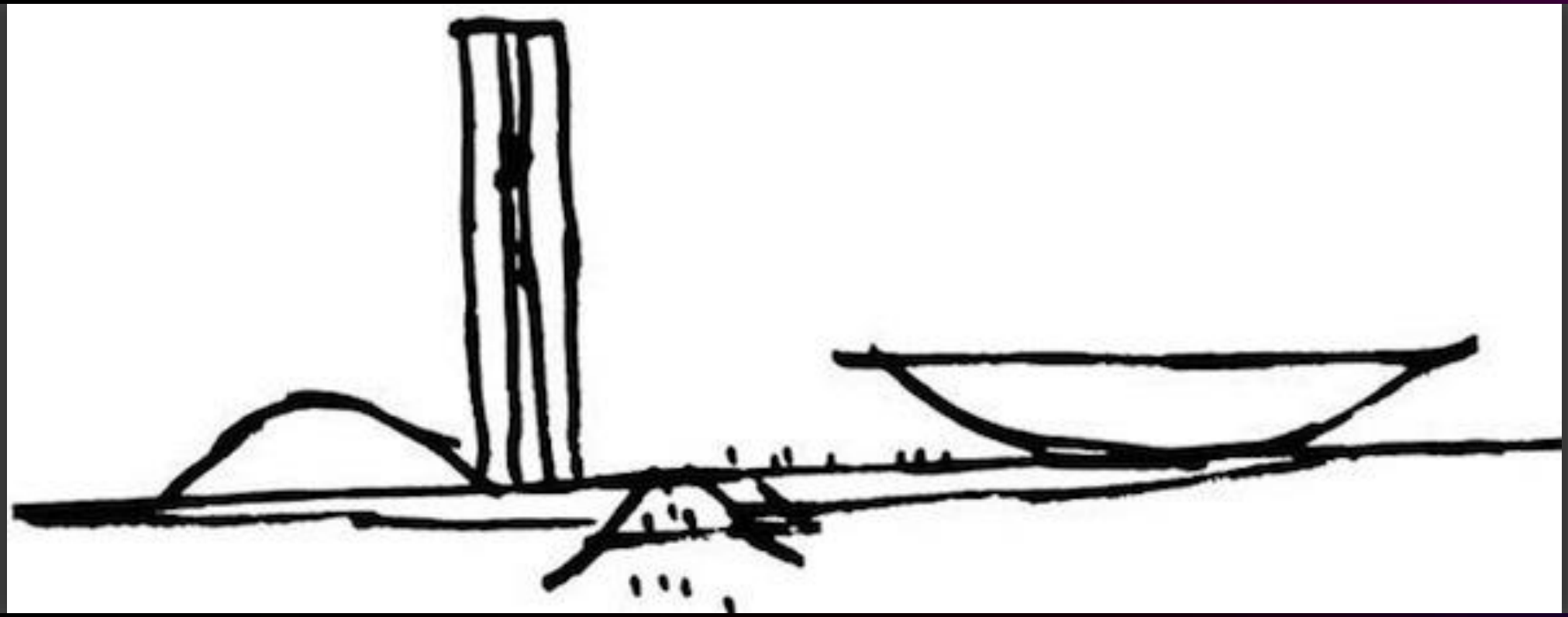

Amazon SNS


AWS Step Functions


AWS AppSync

Thinking like an architect

Famous architects sketch



Oscar Niemeyer, sketch of the Brazilian National Congress

Famous architects sketch

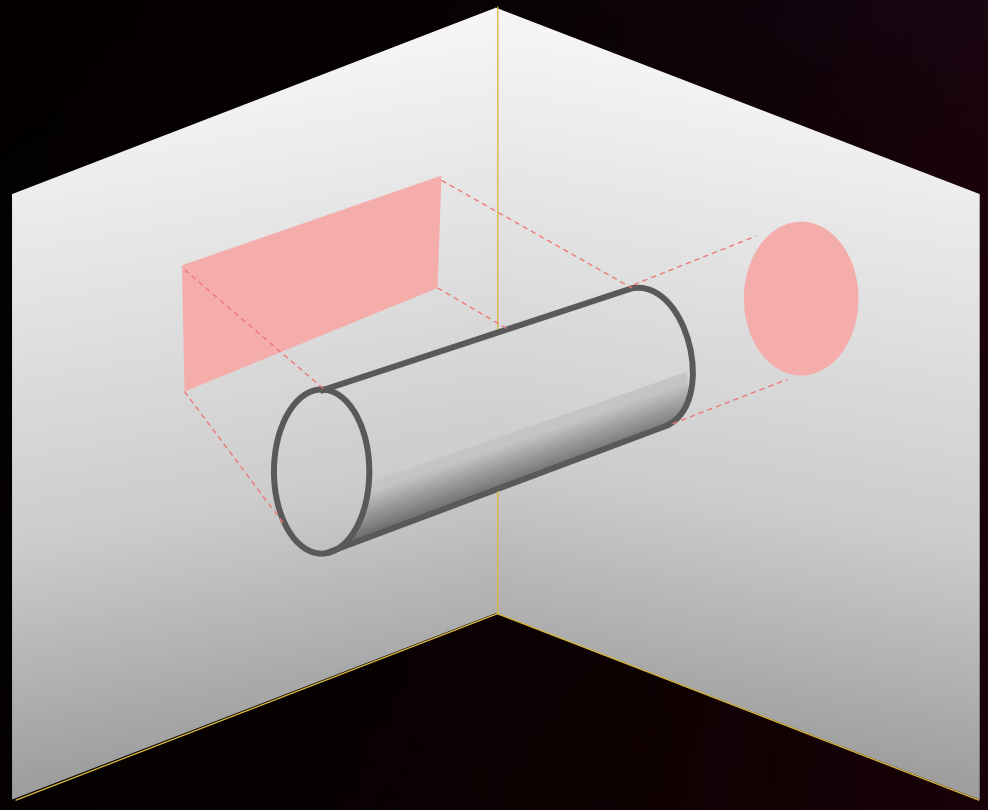
Oscar Niemeyer, sketch of the Brazilian National Congress

Famous architects sketch

Agência Senado

Architects see more dimensions

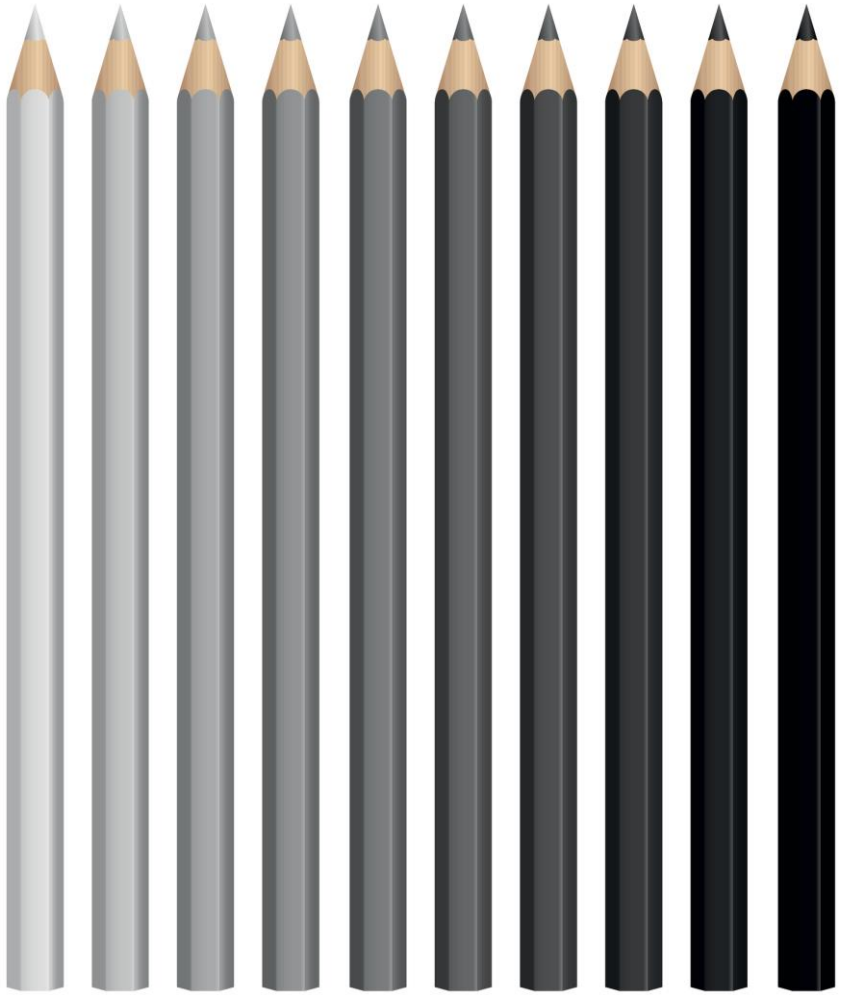
This is a **circle**!



This is a **rectangle**!

Folks, you're **both** right!

Architects see shades of gray



“Architects raise the level of abstraction to deepen everyone’s thinking.”

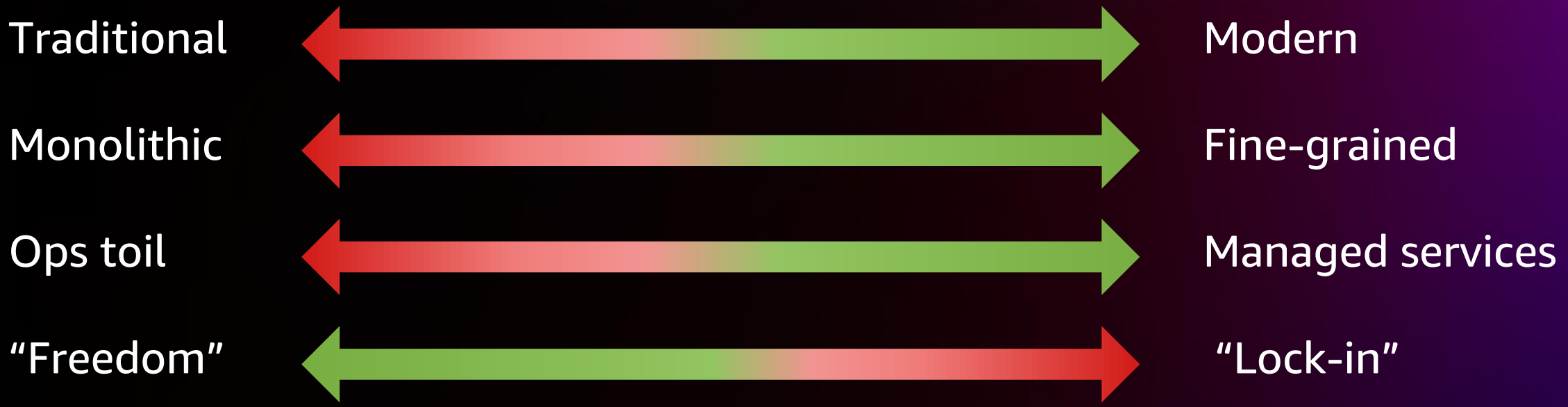
Seeing more cloud dimensions

“Dear Annie, I feel that my platform is too clingy. How can I get the most of out of our relationship while retaining my freedom?”

Freedom Seeker

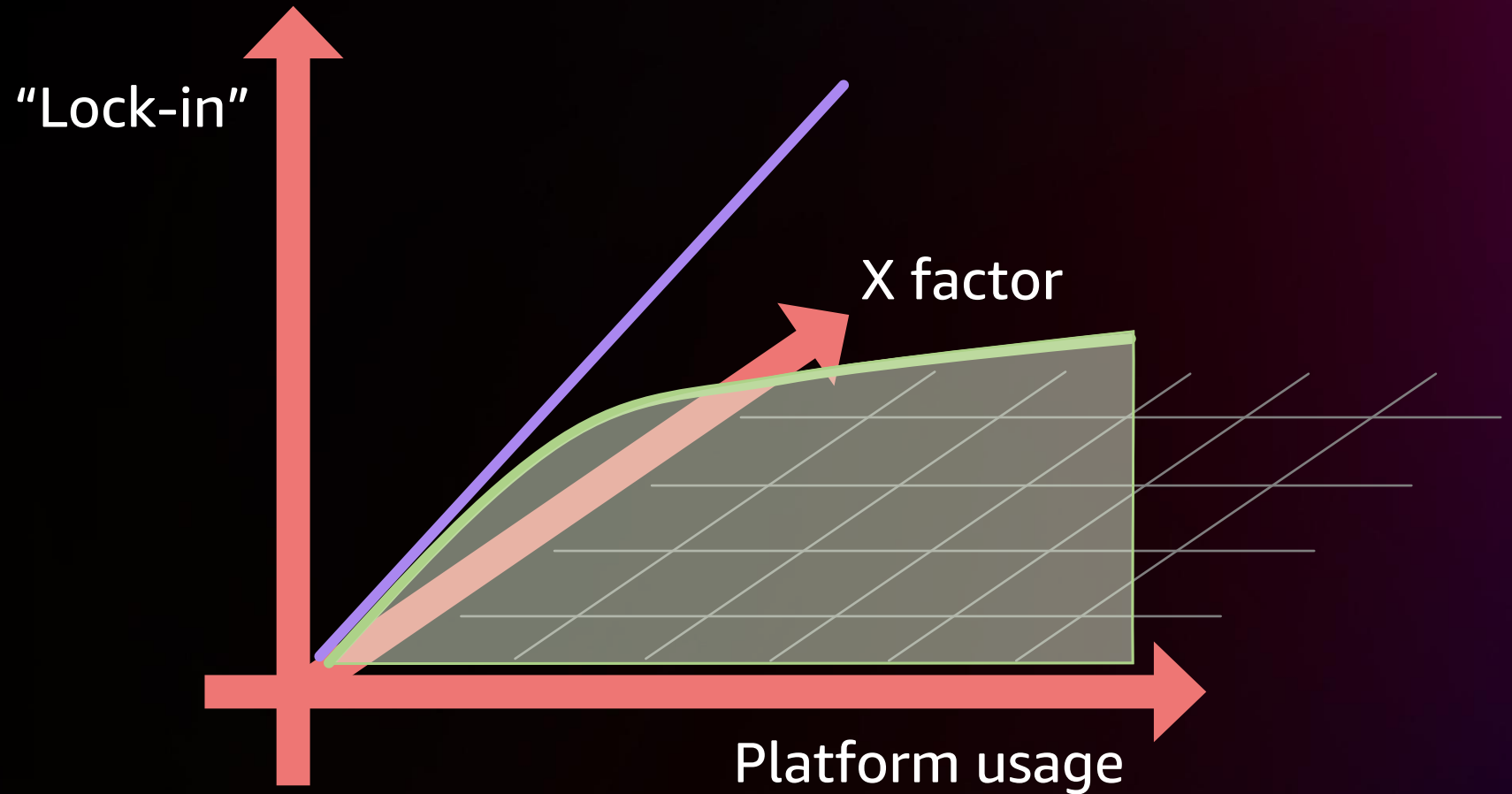
Las Vegas, NV

The single-dimensional view



In the one-dimensional view, there is no happy place

Architects see more dimensions



“Lock-in” has many dimensions

- Vendor
- Product
- Version
- Architecture
- Skills
- Legal
- Mental

Each represents potential switching costs

• Vendor	How much does a switch from vendor A to B cost?
• Product	How much does a migration to another product cost?
• Version	How difficult to upgrade?
• Architecture	How difficult to refactor?
• Skills	What learning paths and how much cognitive load?
• Legal	Can I renegotiate or influence the regulator?
• Mental	How to shed old assumptions?

https://martinfowler.com/articles/oss-lockin.html

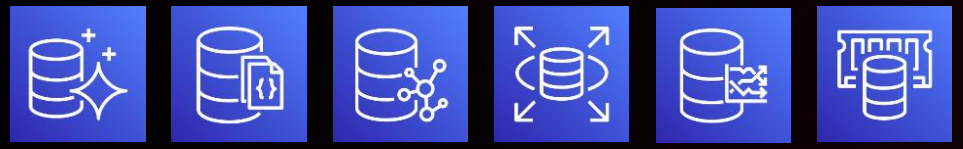
Seeing more dimensions gives us new insights

Open-source software reduces vendor lock-in but retains most other forms, specifically product and architecture lock-in

Mental lock-in is the most subtle but also the most difficult kind to overcome

Changing providers – Service mappings

Vendors appear to offer comparable services

	"Over here"	"Over there"
Virtual machine		
Serverless function		
Message queue		
NoSQL database		
		

There’s a lot behind one service icon

Amazon EventBridge

Build event-driven applications at scale across AWS, existing systems, or SaaS applications

Features (straight from the docs)

- Global endpoints
- API destinations
- Archive and replay events
- AWS Glue Schema Registry
- Fully managed and scalable event bus
- SaaS integration
- Over 100 built-in event sources and targets
- Decoupled event publishers and subscribers
- Event filtering
- Reliable event delivery
- Automatic response to operational changes in AWS services
- Scheduled events
- Monitoring and auditing
- Security and compliance

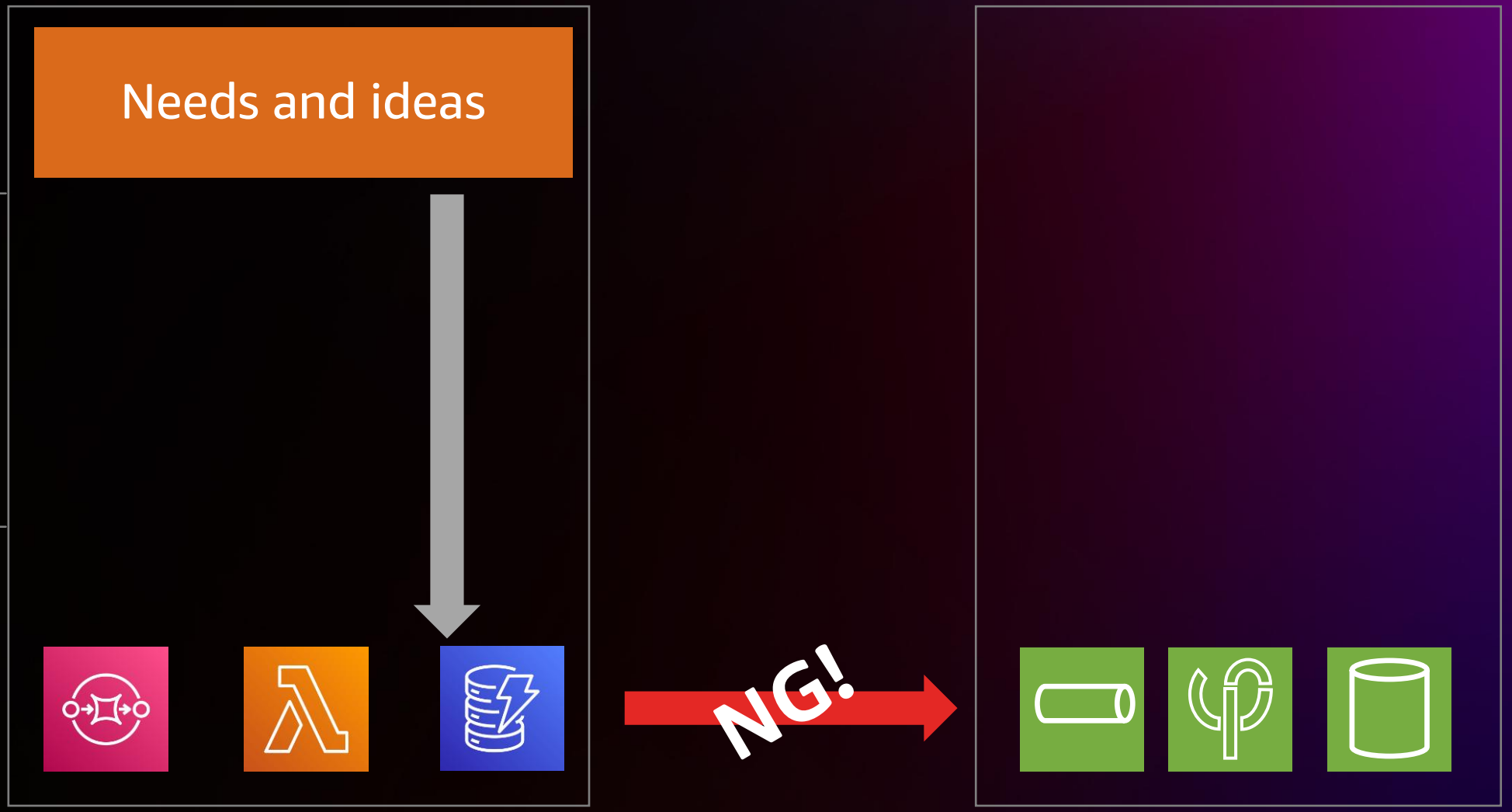
Plus, services don’t stand in isolation; they’re part of an integrated platform

Service mappings don’t work

“Requirements”

Design time

Runtime



“Over here”

“Over there”

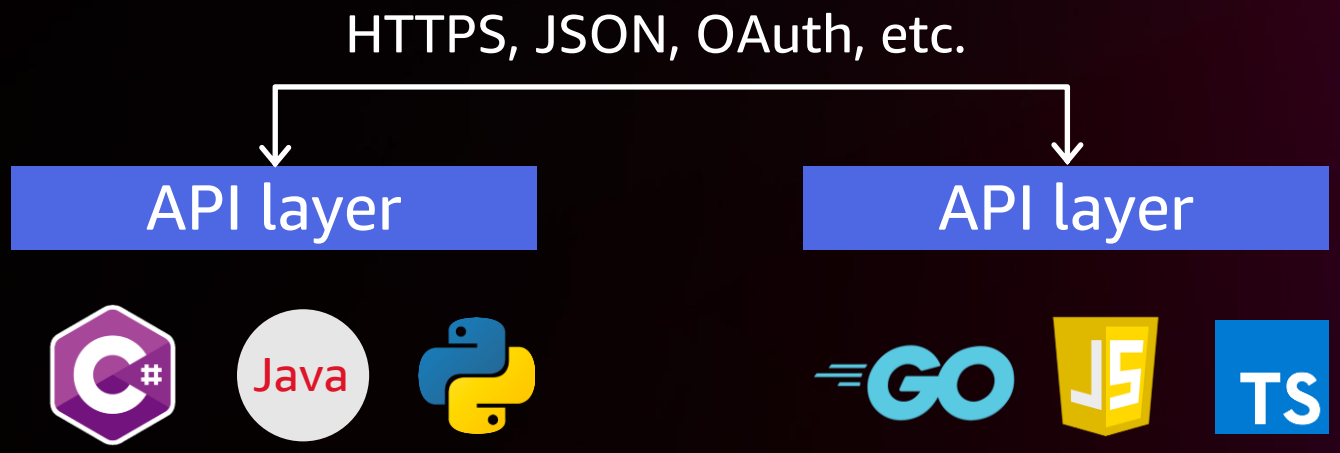
Abstraction layers

“We can solve any problem by introducing an extra level of indirection.”

David J. Wheeler

The fundamental theorem of software engineering

Agreeing on common elements affords flexibility



“Architects sell options.”

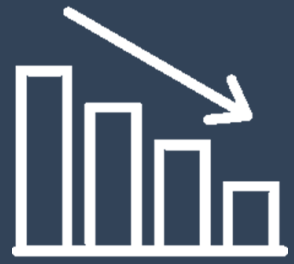
Hohpe, *Software Architect Elevator*

Options aren’t free

Effort

Opportunity cost

Complexity



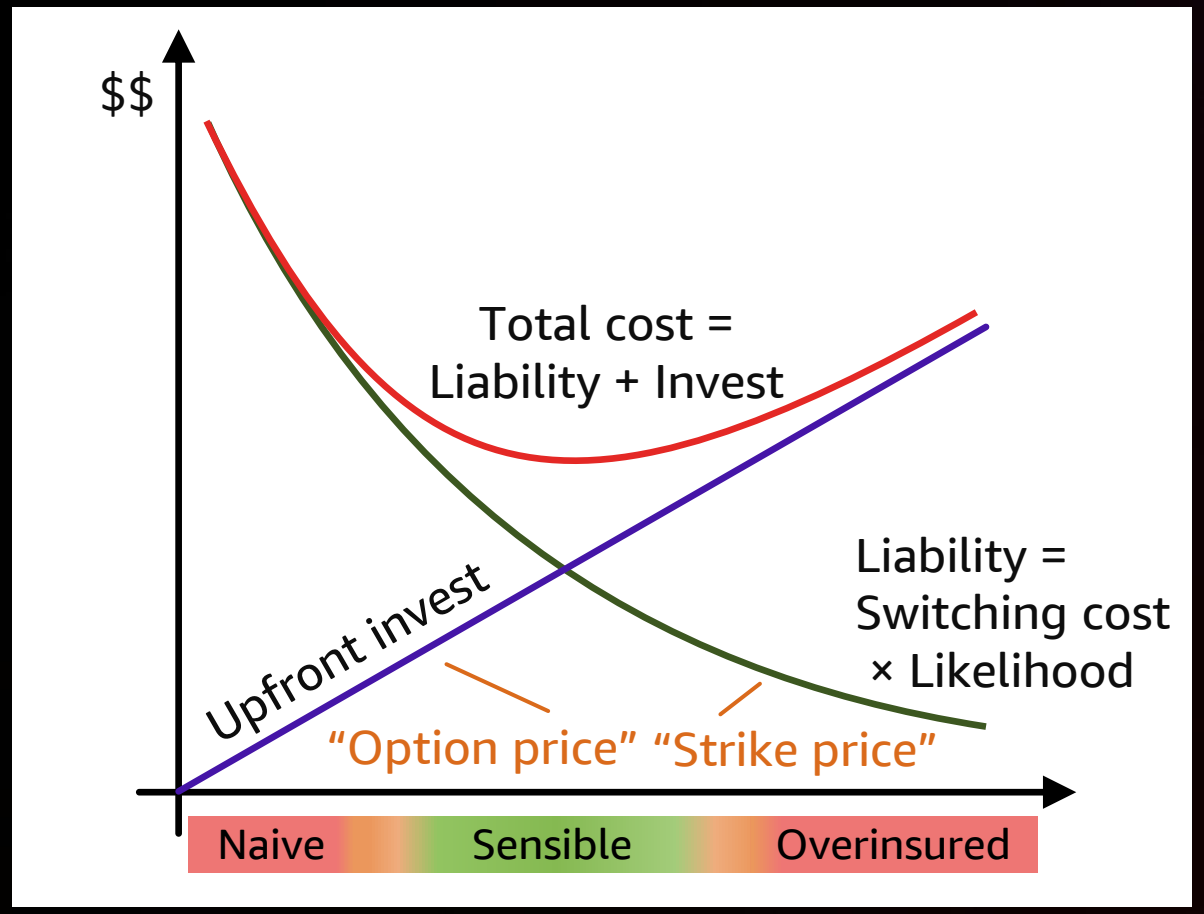
Underutilization

“Excessive complexity is nature’s punishment for organizations that are unable to make decisions.”

Gregor’s Law

https://architectelevator.com/gregors-law/

Options trading: Option price vs. strike price



Source: Hohpe, *Cloud Strategy*

(Simplified)

Total cost =
option price +
strike price x likelihood of use

Zero strike price means it's no longer an option; it's a purchase

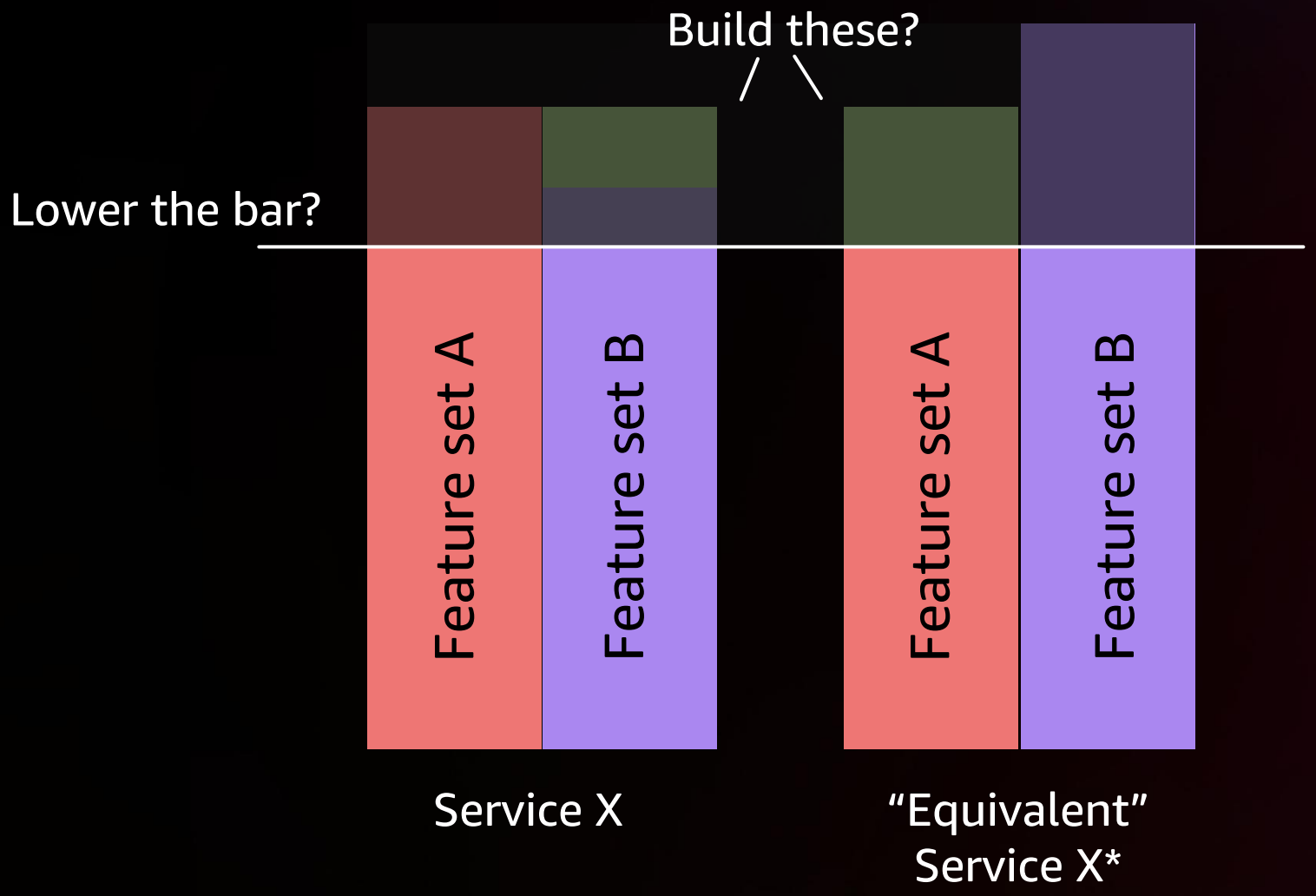
Option price incurs today, payoff is potential and in the future; future money is worth less than today's money

“Fast-moving companies rarely build a complex portability framework. The cost to them (in time and expense) would be far too high.”

“We need something like SQL”

- Developed by one vendor for a specific product
- Designed to make interacting with relational databases easier, not to achieve portability
- Rests on a solid computational model: relational algebra
- 587-page specification (ANSI X3.135-1992, ISO 9075:1992) – now in parts
- Virtually every vendor includes proprietary extensions
- Improves vendor and skill switching dimensions, but not versions, architecture, commercial
- Doesn’t shield you from underlying runtime considerations; you can’t abstract away the laws of physics or failure with a logical layer

The lowest common denominator problem



Also to keep in mind

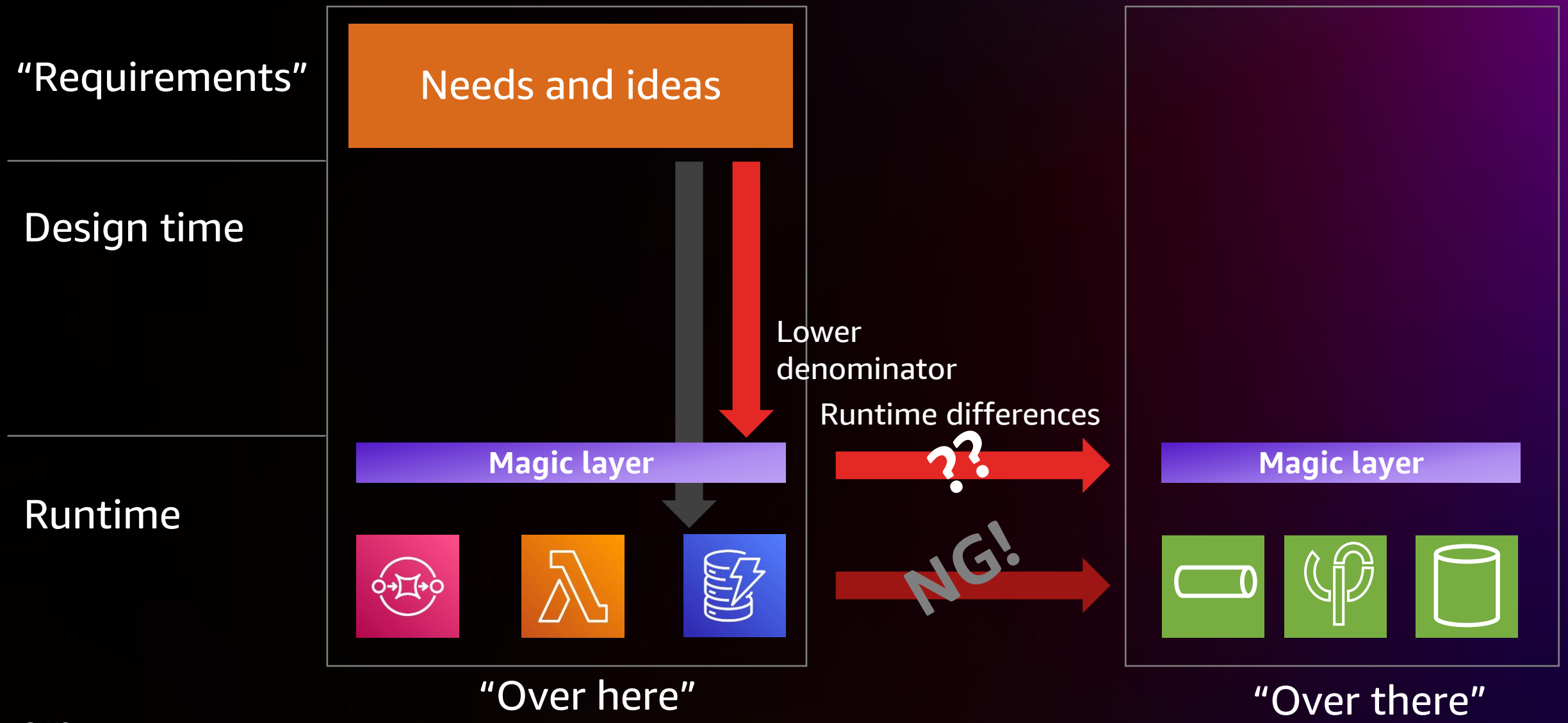
- Runtime characteristics
- Security posture
- Operational complexity
- Integration with other services
- Scaling mechanisms
- Pricing model
- Cost
- …

“Beware the Grim Wrapper”

Hohpe, *Platform Strategy*

“A product or framework must help you today and not just possibly some time in the future.”

Bottom-up abstractions don’t really work, either



The X factor: Velocity

“Portability is important, but only if you have a successful product first.”

Velocity: Good now and in case of migration

Eliminate friction

Continuous integration

Relentless automation

Autonomy, local decisions

DevOps

Reduce inventory

Continuous delivery

Limit work in progress

“Always ready to ship”

Lean

Avoid low-value work

Limit dependencies

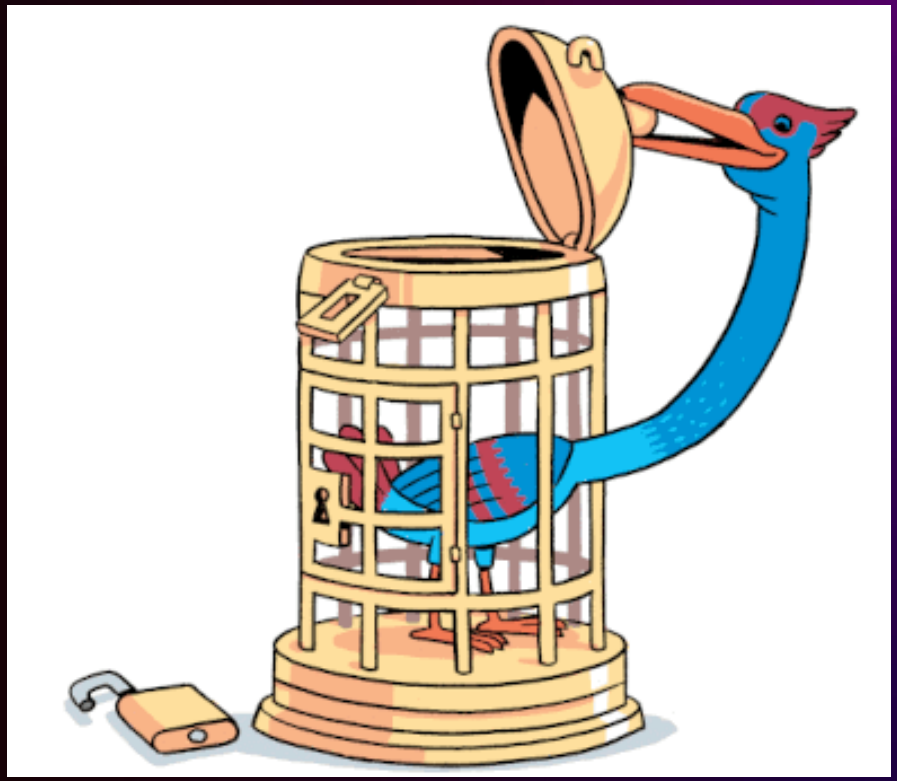
Optimize for value delivered

Gather feedback and iterate

Agile

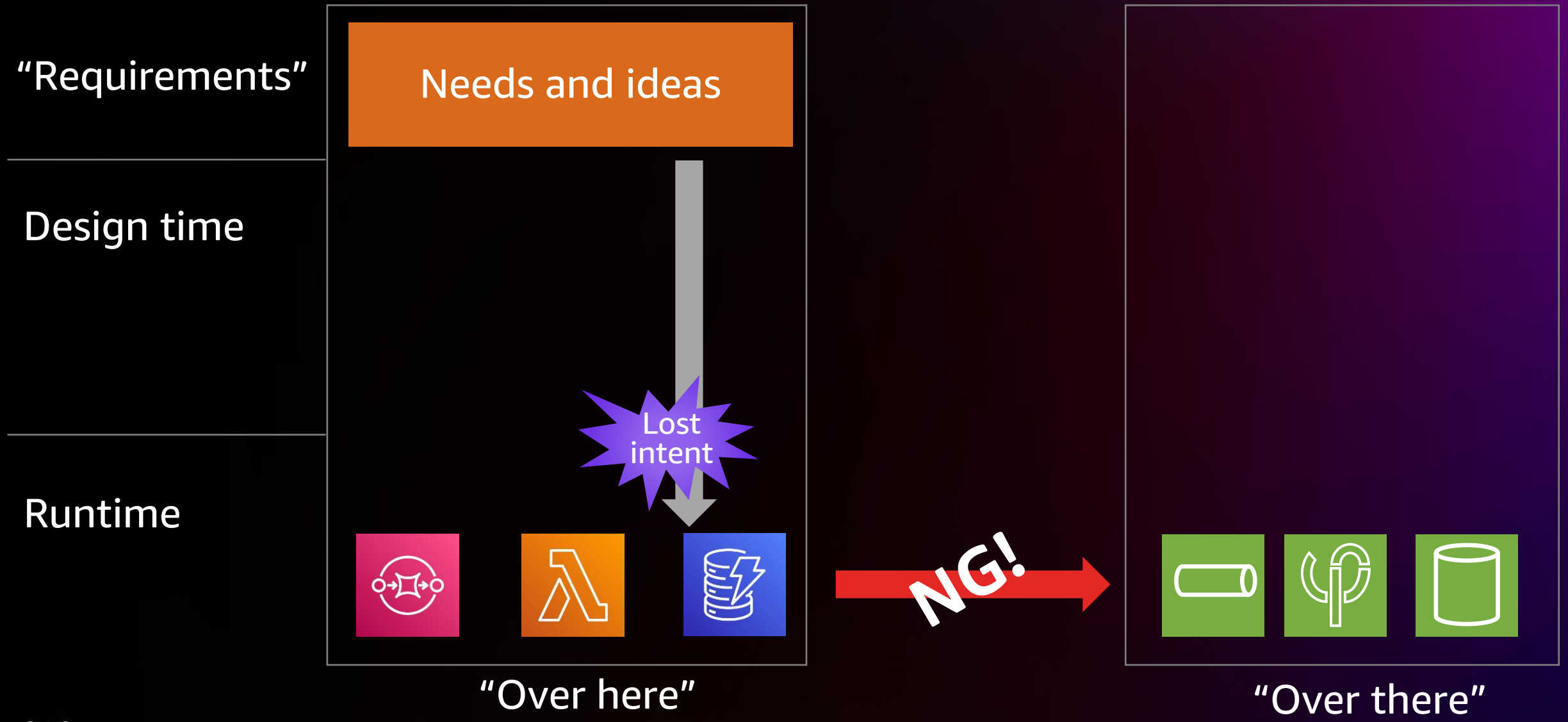
Switching costs isn’t just a function of the services you use; it’s also not an “infrastructure” concern

It’s largely a function of how you work; increasing velocity greatly reduces your switching costs!



Preserving design intent with patterns

The problem with service mappings

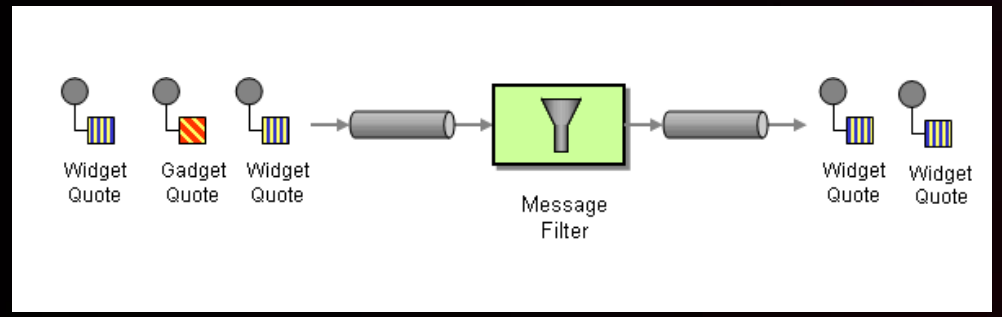


What's your intent behind using a service?

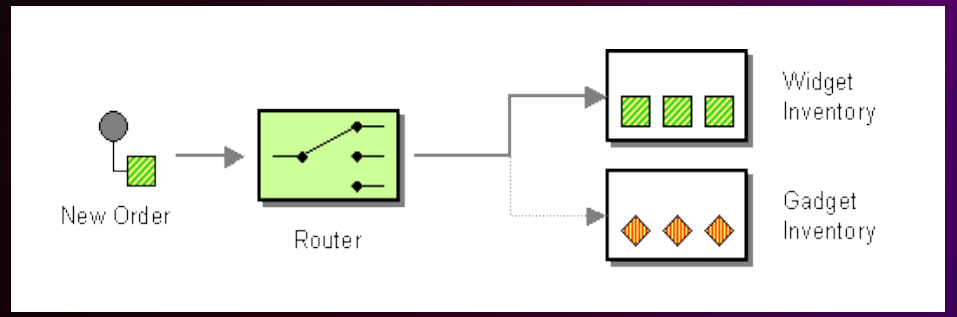
Amazon EventBridge

?

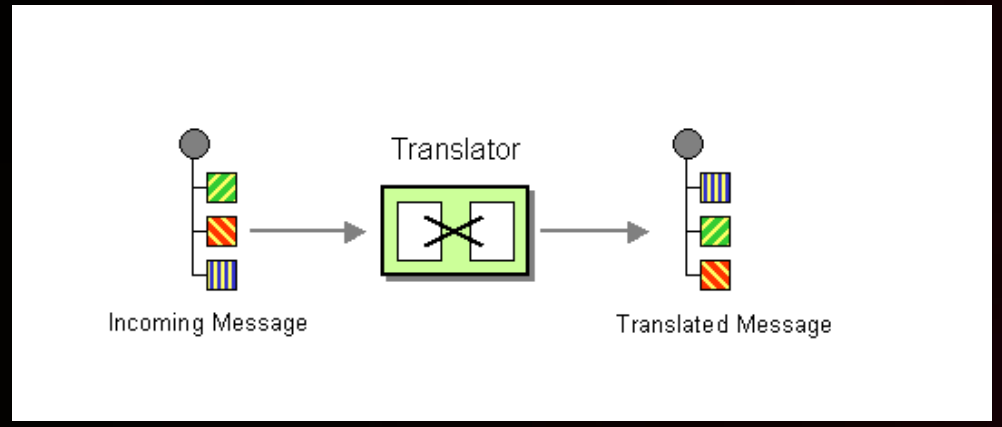
Message Filter



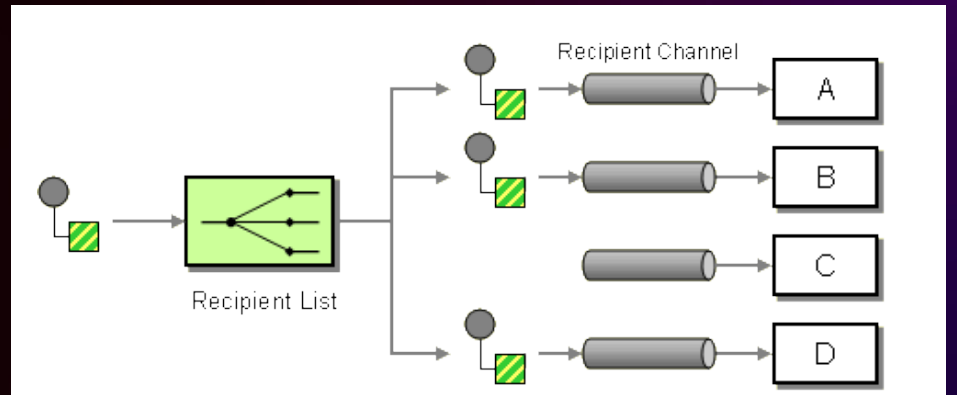
Content-Based Router



Message Translator



Recipient List

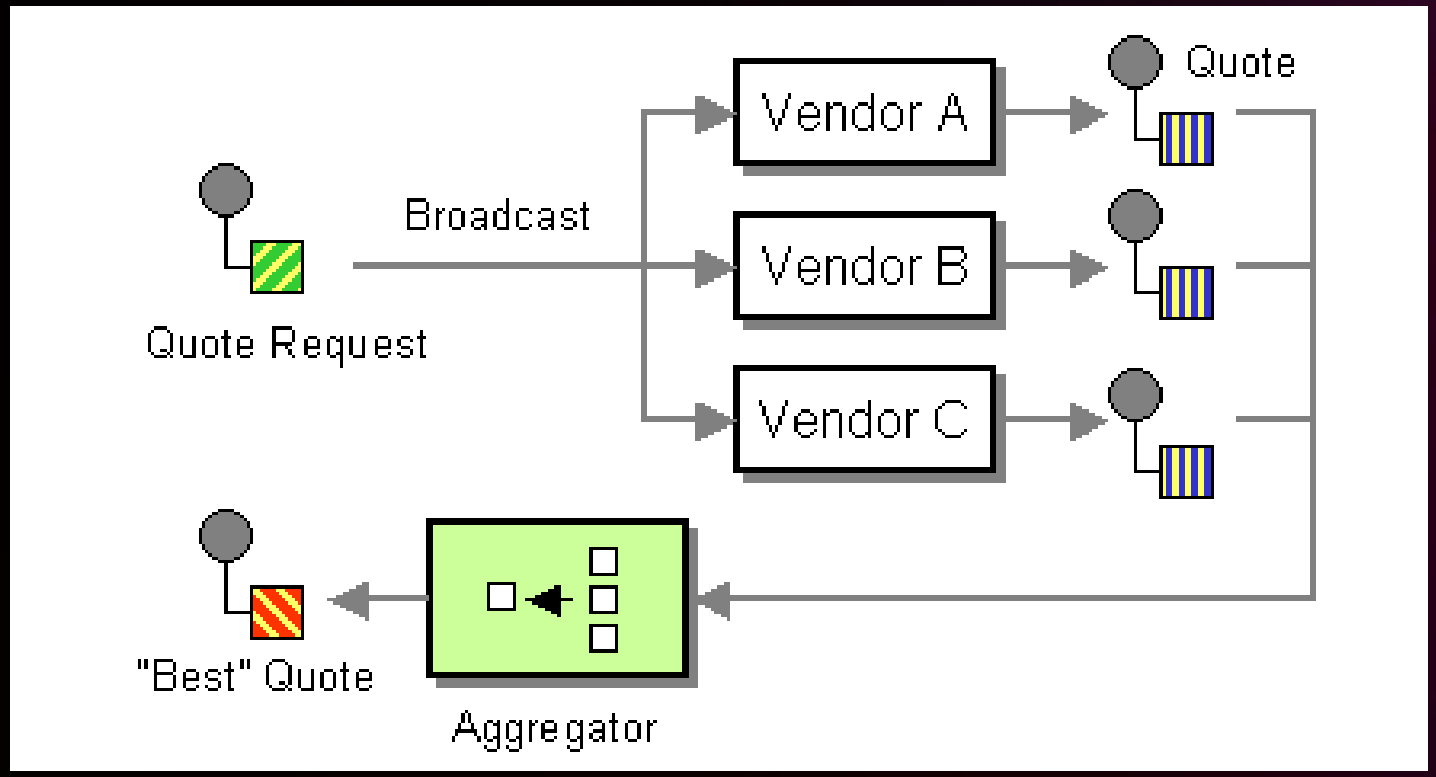


https://www.enterpriseintegrationpatterns.com/

“Thinking only in platform services loses your application’s design intent and mentally locks you in.

Thinking in design patterns retains the intent.”

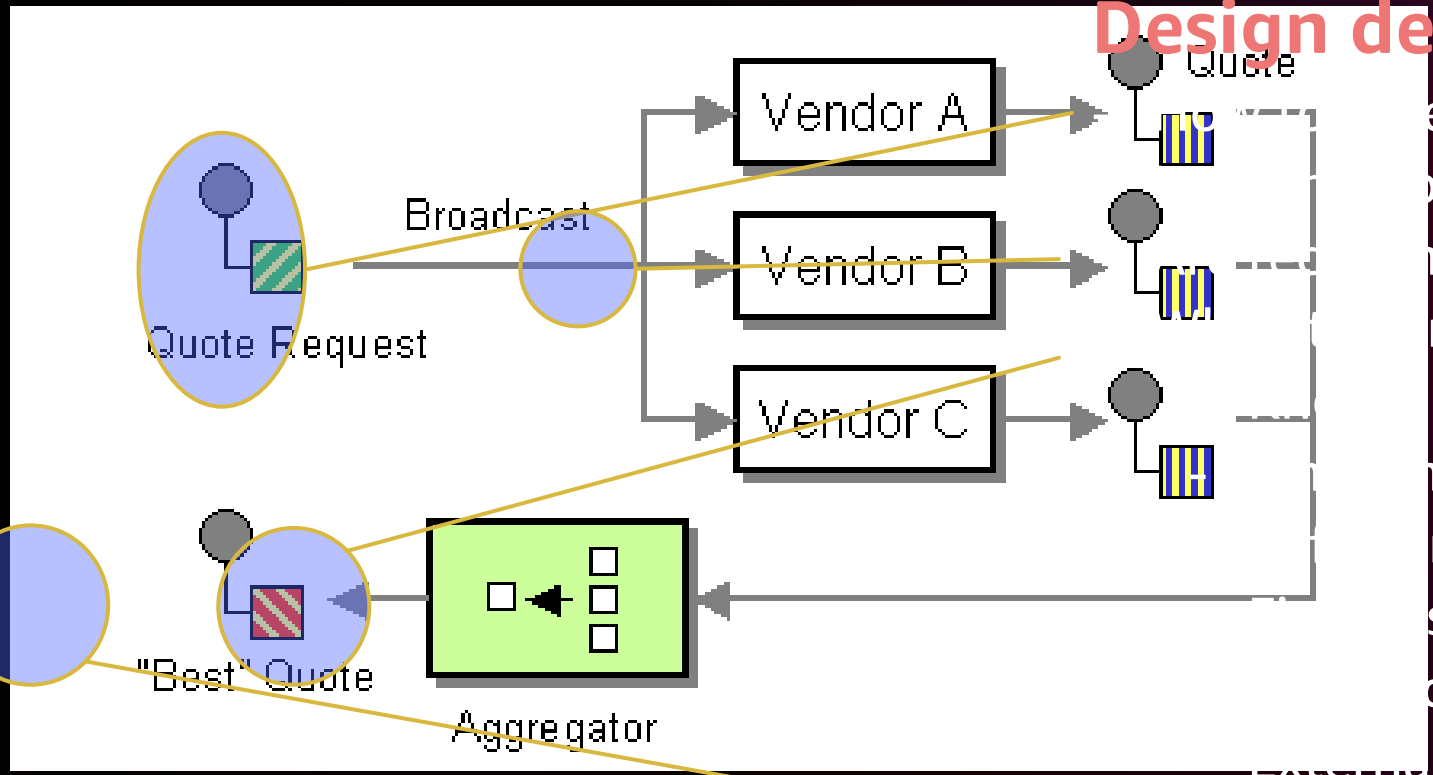
A central pattern: Scatter-Gather



"How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?"

https://www.enterpriseintegrationpatterns.com/patterns/messaging/BroadcastAggregate.html

A central pattern: Scatter-Gather



Design decisions

- ermine set of recipients?
  - er known?
  - nts required to respond?
  - mplete aggregation:
    - number (“wait for all”)
    - mit (“time-out”)
    - nt number of responses
    - st (1 is sufficient)
    - ce of favorable response
    - l event (“gavel drops”)
- How to combine responses
  - Concatenate
  - Select “best”
  - Combine (sum, avg.)

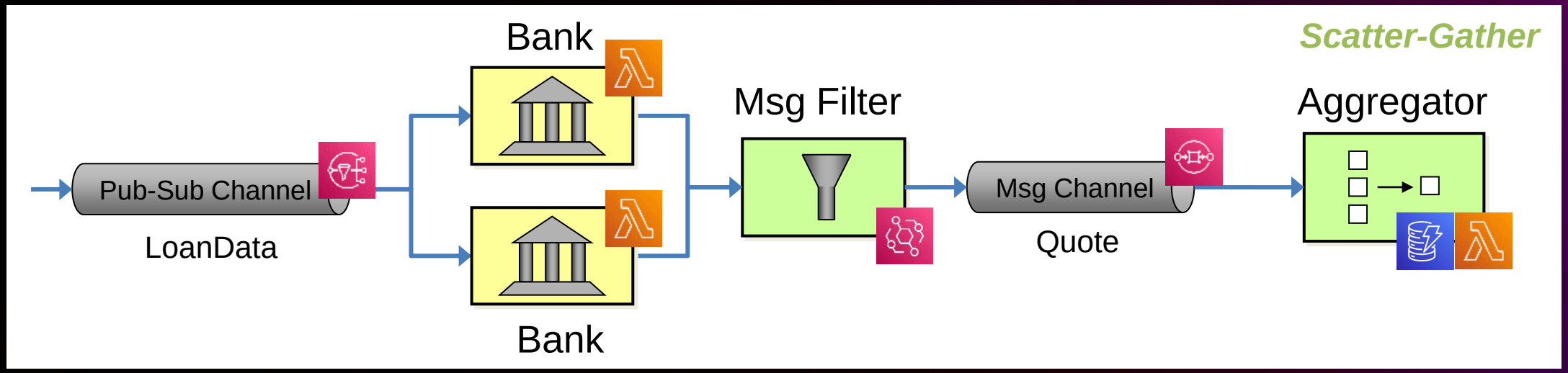
None of these decisions depend on the service selection; they can also be suitably discussed with business users

“You might be overlooking critical design decisions in favor of service selection.

Capturing those decisions as patterns reduces your switching cost.”

https://architectelevator.com/architecture/important-decisions/

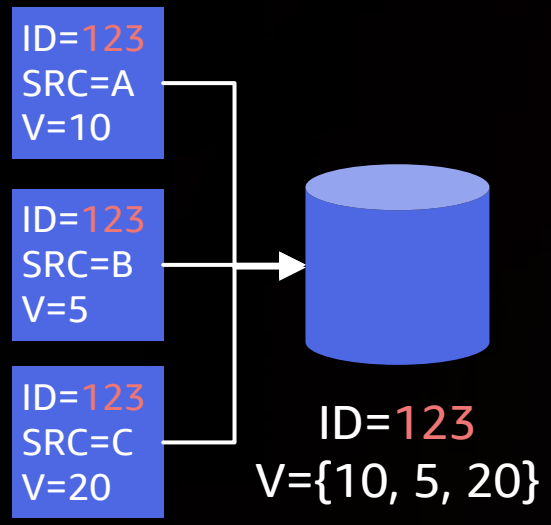
Separate vocabulary by layer



Business domain	Integration patterns	AWS resources
• Bank • Loan request • Loan broker • Mortgage quote	• Message Filter • Content Filter • Aggregator • Publish-Subscribe	• AWS Lambda function • Amazon SQS/Amazon SNS queues and topics • AWS Step Functions tasks • Amazon EventBridge rules

Coding patterns

I need to combine several messages




```
TableName: tableName,
Key: { ID: { S: id } },
UpdateExpression:
  "SET #data = list_append(
     if_not_exists(#data, empty_list),
     :v)",
```

```
key = db.key("ID", id)
with db.transaction():
  item = db.get(key)
  if not item:
    item = datastore.Entity(key)
    item['data'] = []
  item['data'].append({ 'Value': v })
  db.put(item)
```

Transaction limit

↓

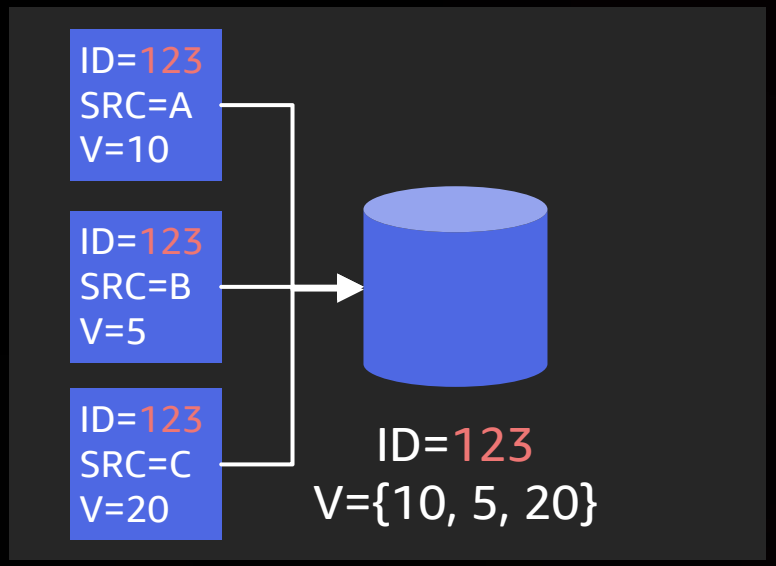
Retries (on queue)

↓

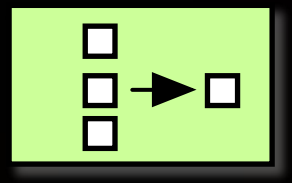
TTL (in function code)

Avoid the issue with composite keys/parent entities?

I am building an Aggregator

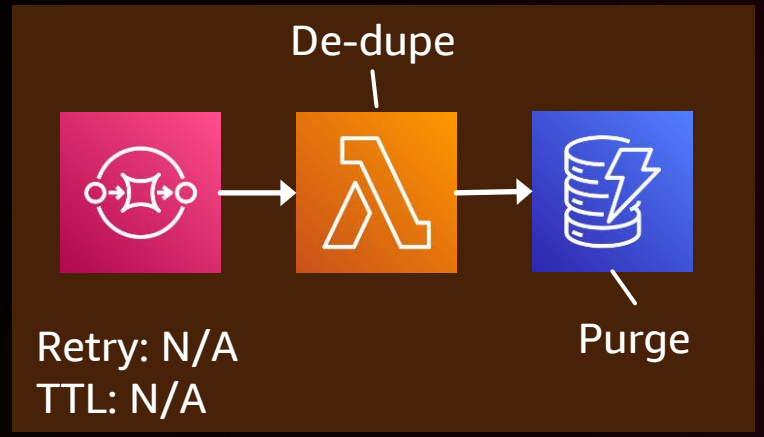



```
new Aggregator(
  corrID = "/ID".
  payload = "/data/V",
  Completeness = Aggregator.Completeness.MinCount(2).
                 Timeout(10),
  Aggregation = Aggregator.Aggregation.Concatenate,
  Duplicates = Aggregator.Duplicates.Ignore,
  MinRetention = 3600
)
```

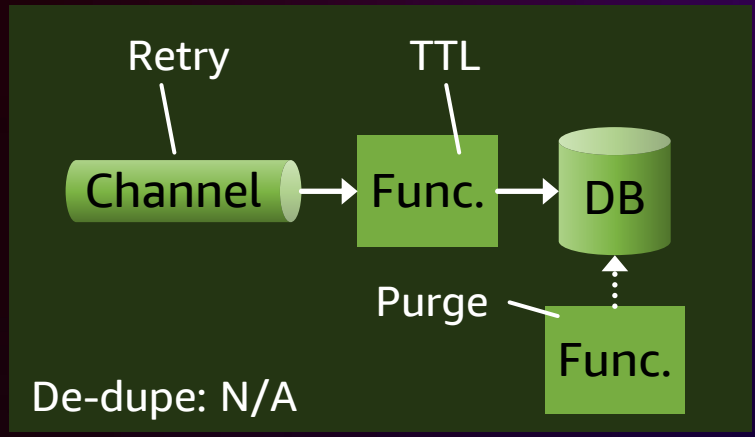


"Use a stateful filter, an **Aggregator**, to collect and store individual messages until a complete set of related messages has been received. Then, the aggregator publishes a single message distilled from the individual messages."

https://www.enterpriseintegrationpatterns.com/Aggregator.html



"Over here"

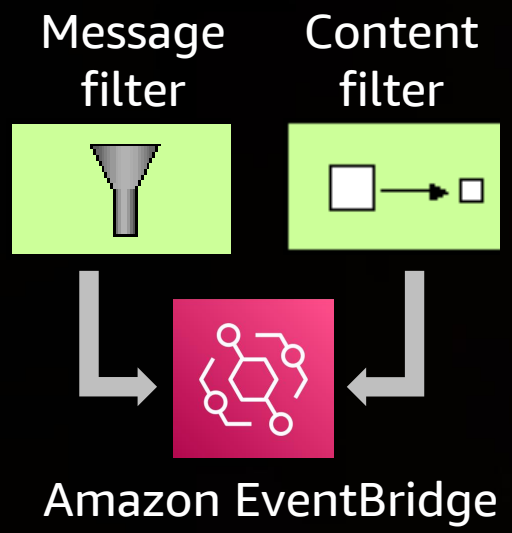


"Over there"

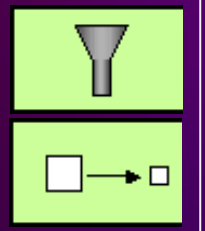
Patterns ~~are~~ design-time abstractions

used to be

"Automation and CDK elevate patterns to programmable deployment-time constructs."



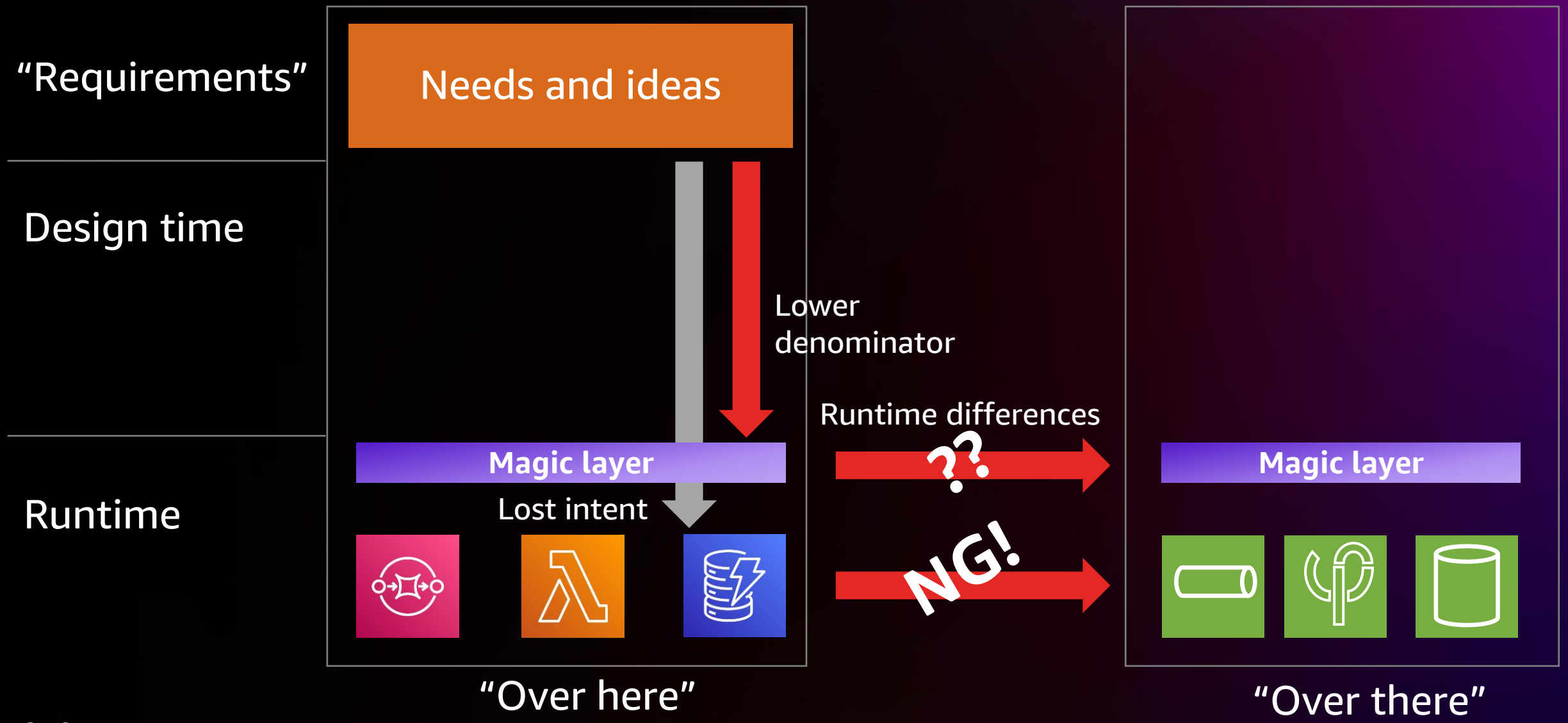

```
nonEmptyQuotesOnly = MessageFilter.fieldExists(this, 'name', 'bankId');
payloadOnly = ContentFilter.payloadFilter(this, 'Filter');

new MessageContentFilter(this, 'FilterMortgageQuotes',
  { sourceEventBus: mortgageQuotes, targetQueue: quotesQueue,
    messageFilter: nonEmptyQuotesOnly, contentFilter: payloadOnly });
```



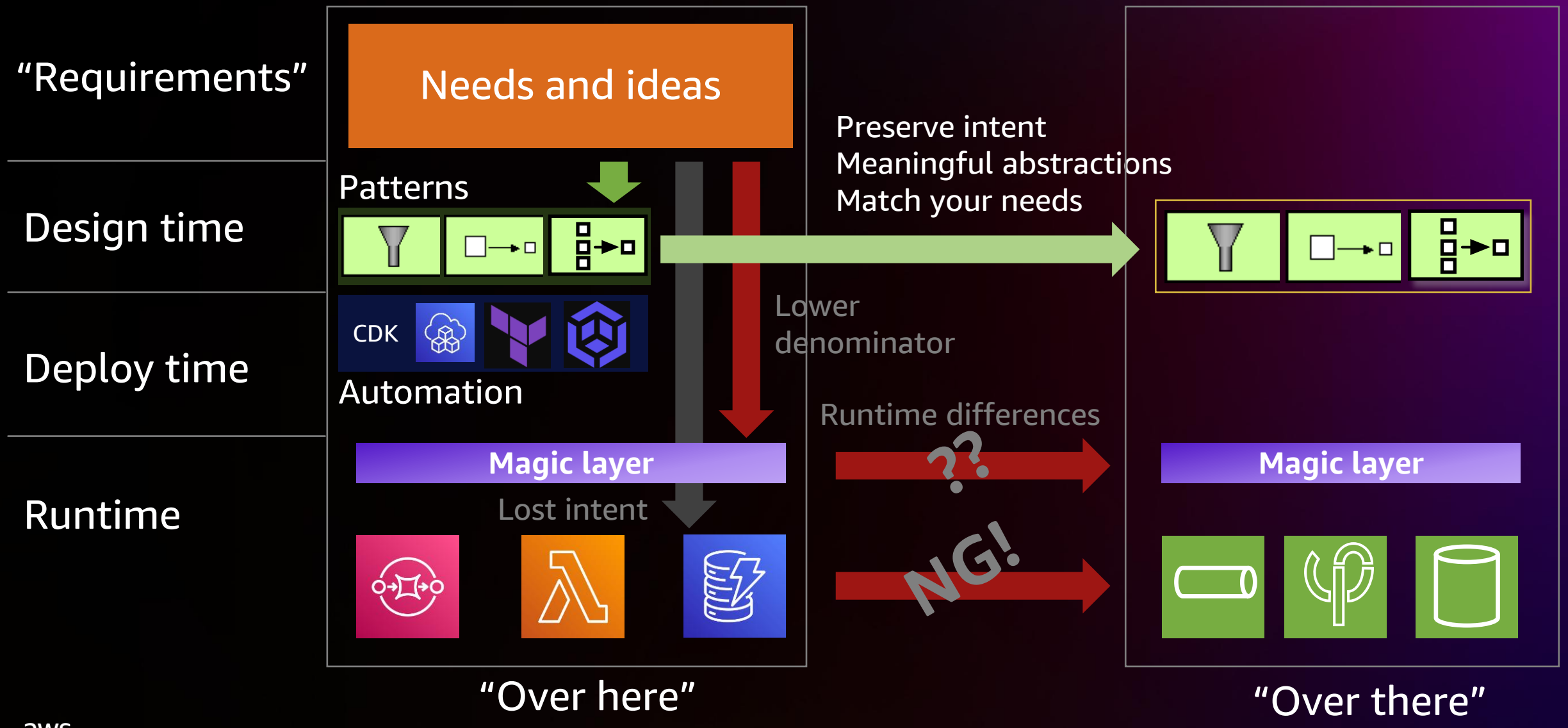
```
class MessageContentFilter extends Construct {
  public readonly eventPattern: EventPattern;
  constructor(scope: Construct, id: string, props: MessageContentFilterProps) {
    super(scope, id);
    const rule = new Rule(scope, id + 'Rule',
      { eventBus: props.sourceEventBus, ruleName: id + 'Rule' });
    rule.addEventPattern(props.messageFilter.eventPattern);
    rule.addTarget(new targets.SqsQueue(props.targetQueue,
                   {message: prop.contentFilter.ruleTargetInput}));
  }
}
```

https://github.com/aws-samples/aws-cdk-loan-broker

Rethinking abstractions and switching costs



Rethinking abstractions and switching costs



Rethinking abstractions and switching costs

The cloud makes us rethink everything

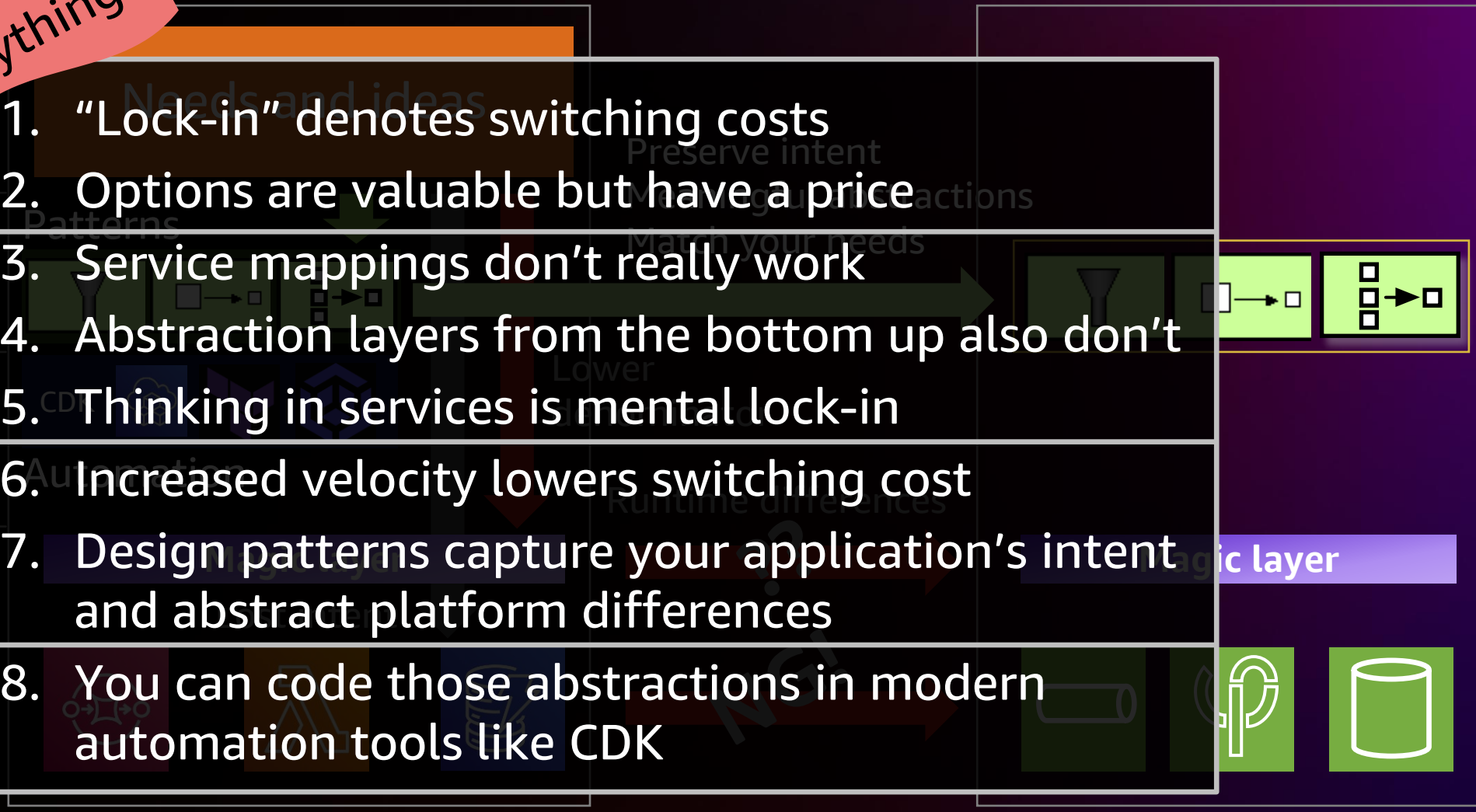
"P

Design time

Deploy time

Runtime

1. "Lock-in" denotes switching costs
2. Options are valuable but have a price
3. Service mappings don't really work
4. Abstraction layers from the bottom up also don't
5. Thinking in services is mental lock-in
6. Increased velocity lowers switching cost
7. Design patterns capture your application's intent and abstract platform differences
8. You can code those abstractions in modern automation tools like CDK



"Over here"

"Over there"

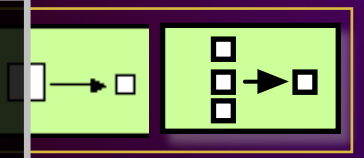
Rethinking abstractions and switching costs

The cloud makes us rethink everything

Design time	1.	
	2.	
	3.	
	4.	o don't
Deploy time	5.	
	6.	
	7.	intent
Runtime	8.	ic layer

Using design patterns and abstractions, plus good software delivery discipline with automation, is the best way to build better applications **and** keep your switching costs low, especially for modern cloud applications

automation tools like CDK



"Over here"

"Over there"

Want more?

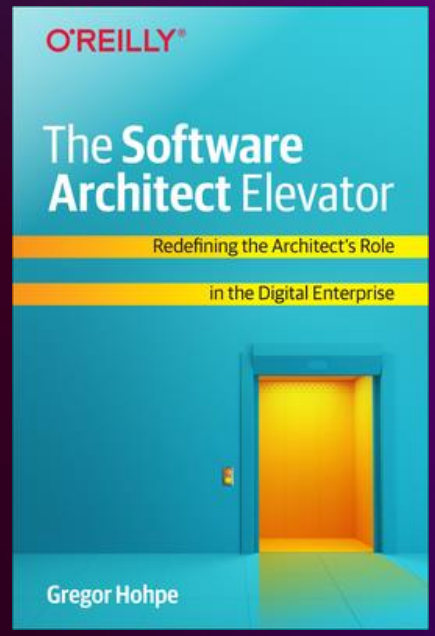
Architect Elevator Blog

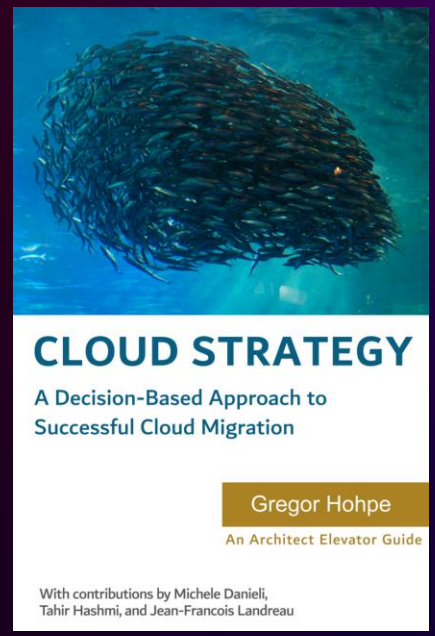
https://ArchitectElevator.com

- “Multi-cloud: From Buzzword to Decision Model”
- “Concerned about Serverless Lock-in? Consider Patterns!”
- “Good abstractions are obvious but difficult to find, even in the cloud”

www.EnterpriseIntegrationPatterns.com

- Loan broker on AWS Serverless
- Serverless automation with AWS CDK
- Porting a serverless application





Thank you!

Gregor Hohpe
@ghohpe
www.linkedin.com/in/ghohpe

Please complete the session survey in the **mobile app**