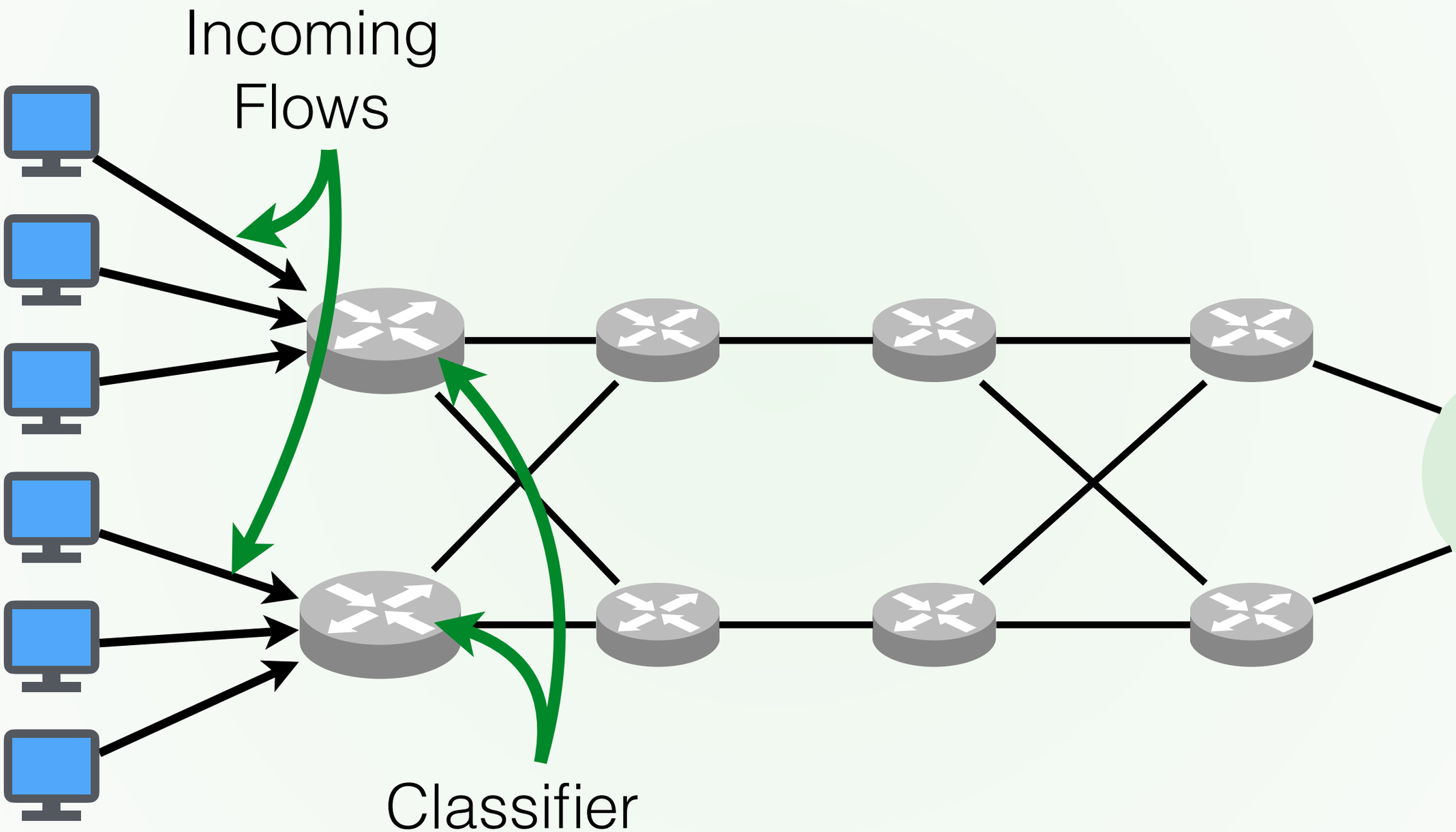
Concise Encoding of Flow Attributes in SDN Switches

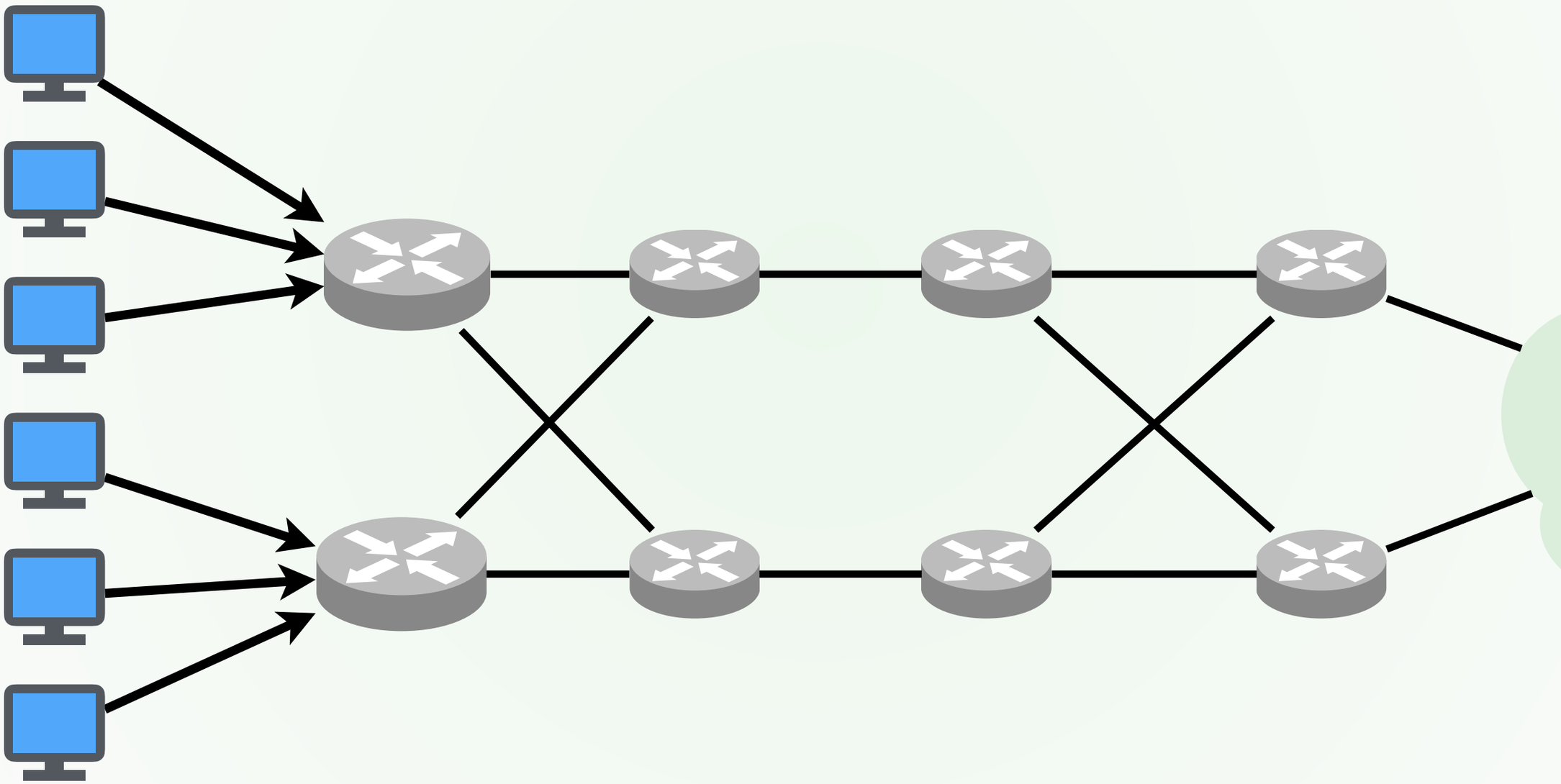
Robert MacDavid*, Rüdiger Birkner†, Ori Rottenstreich*,
Arpit Gupta*, Nick Feamster*, Jennifer Rexford*

*Princeton University, †ETH Zürich

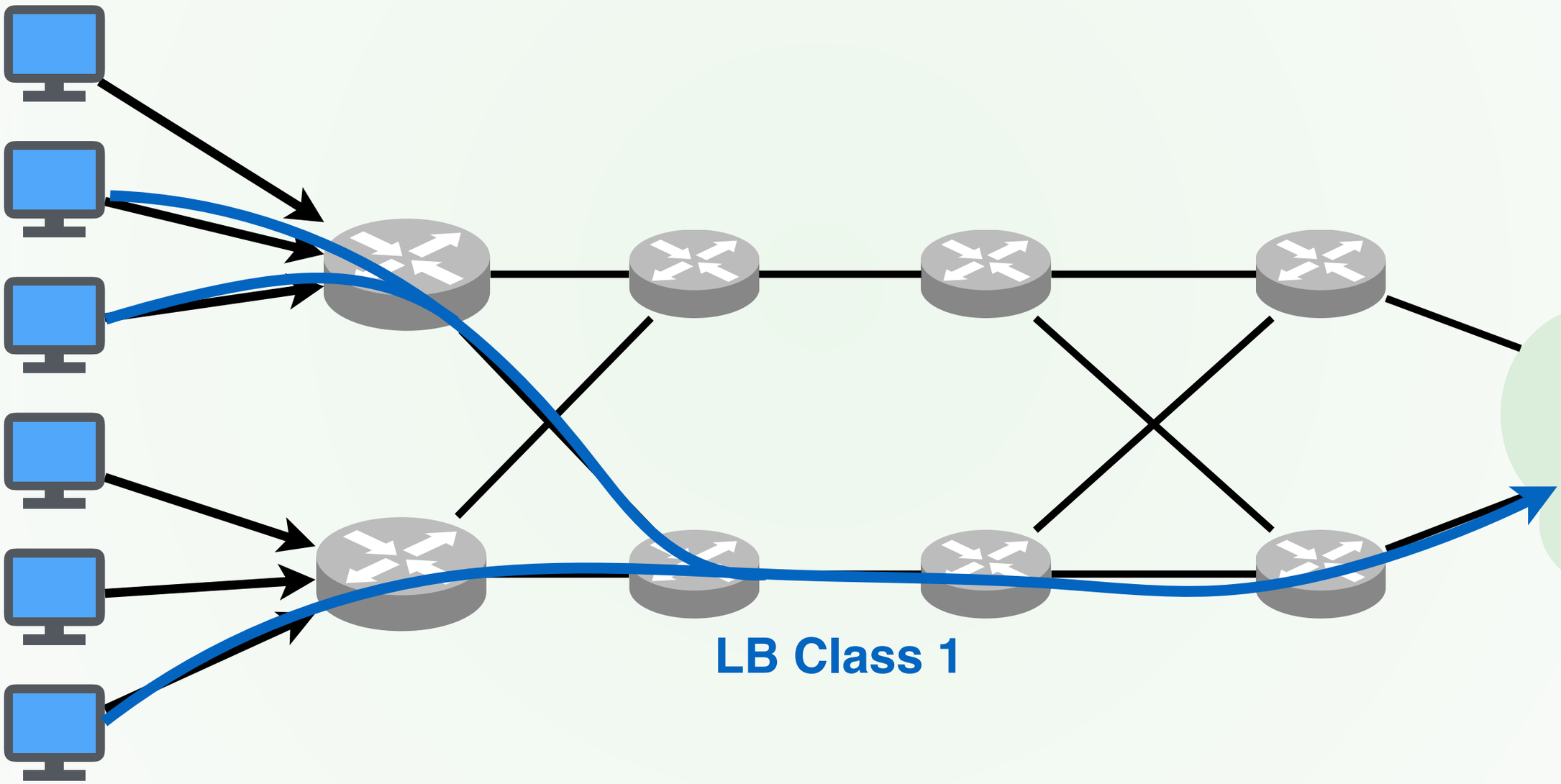
Motivation



Motivation - Load Balancing

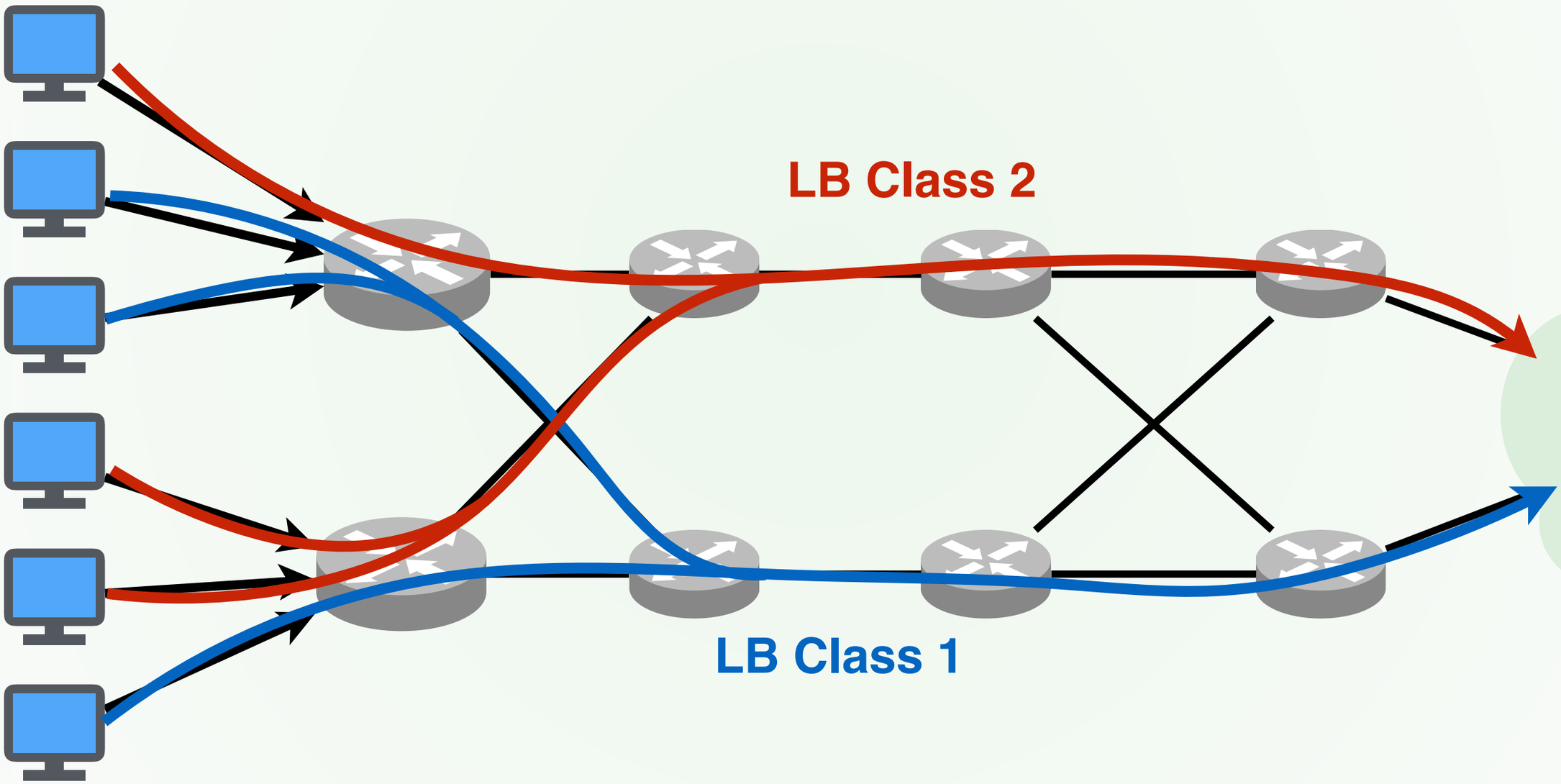


Motivation - Load Balancing

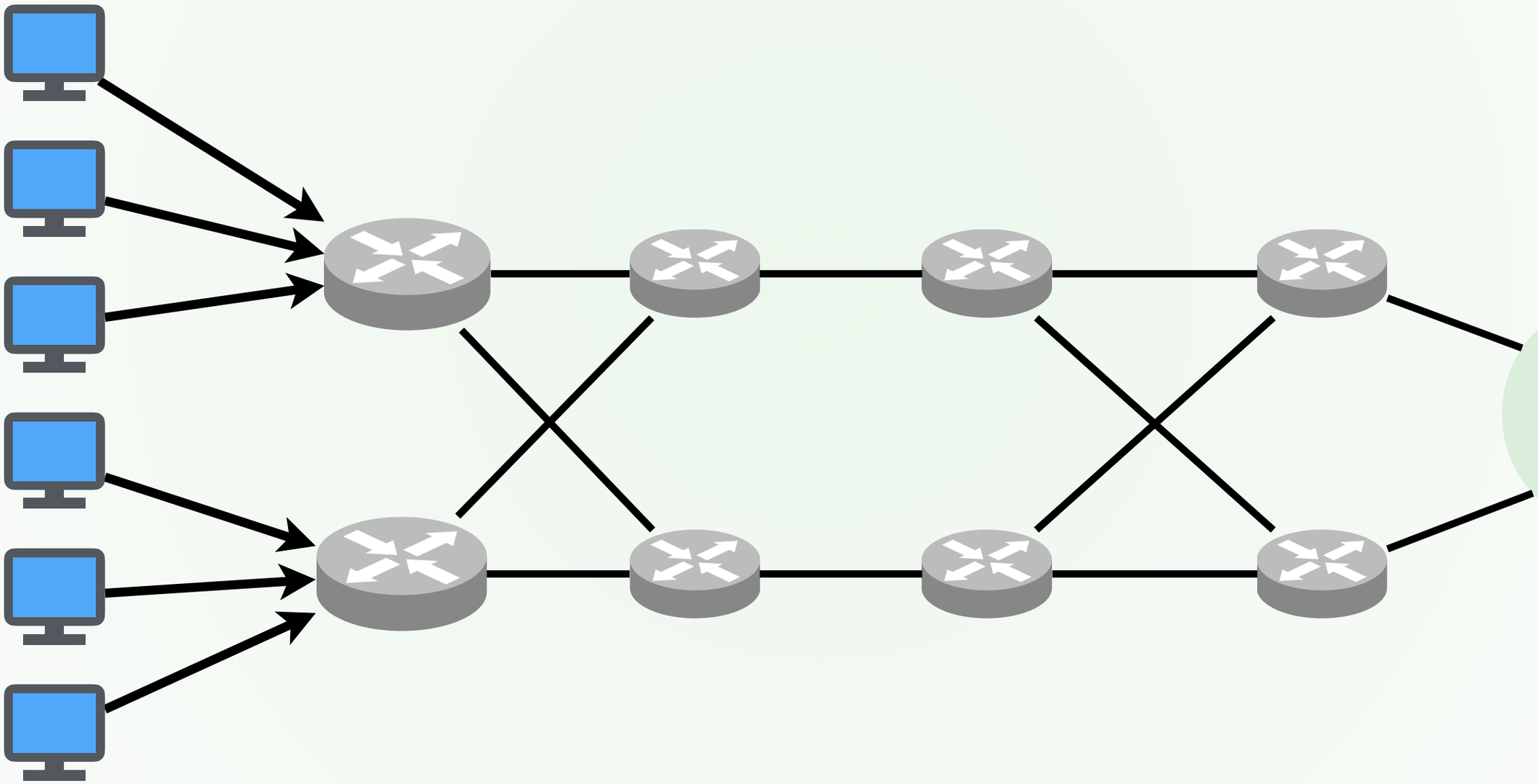


LB Class 1

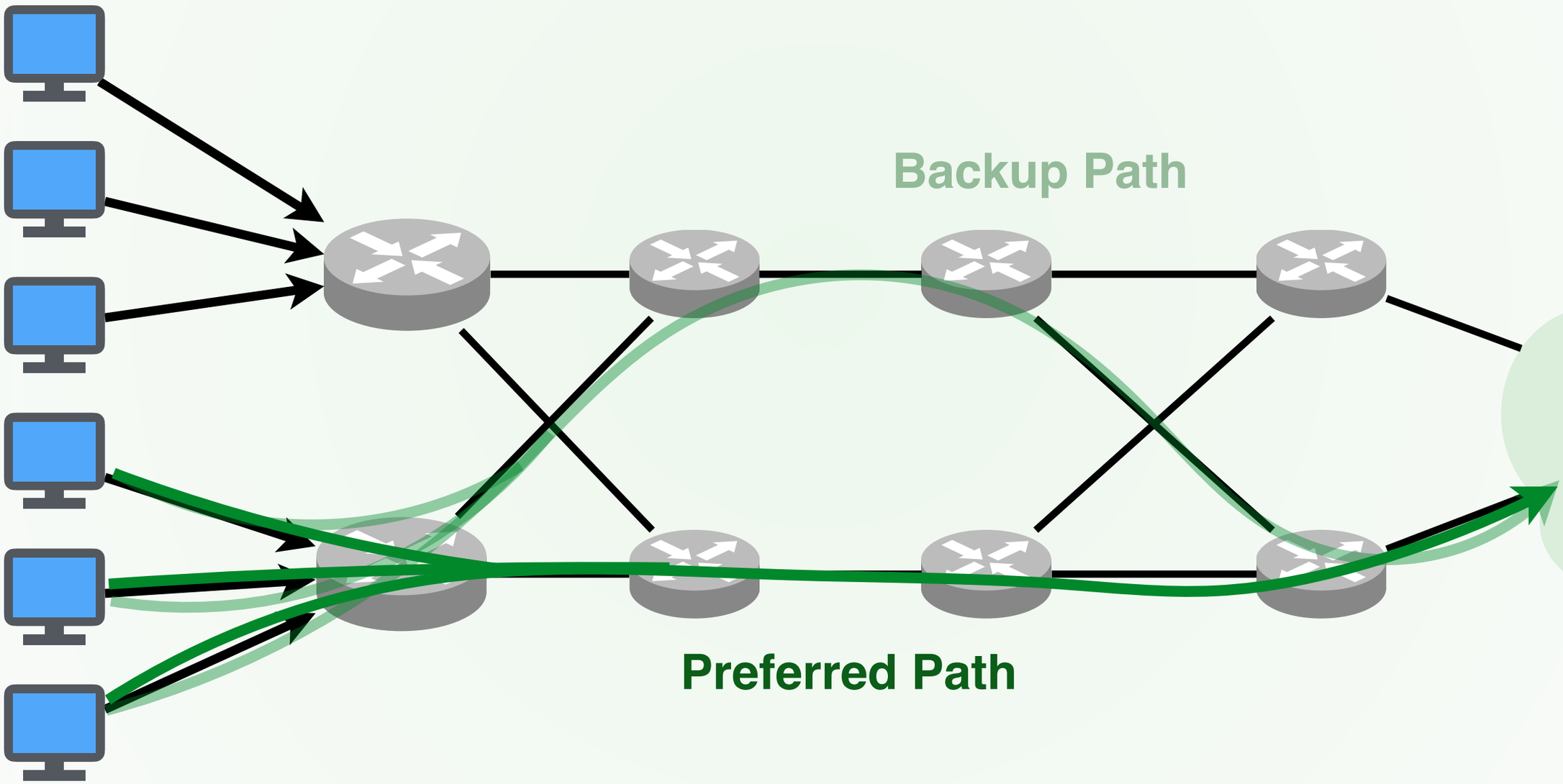
Motivation - Load Balancing



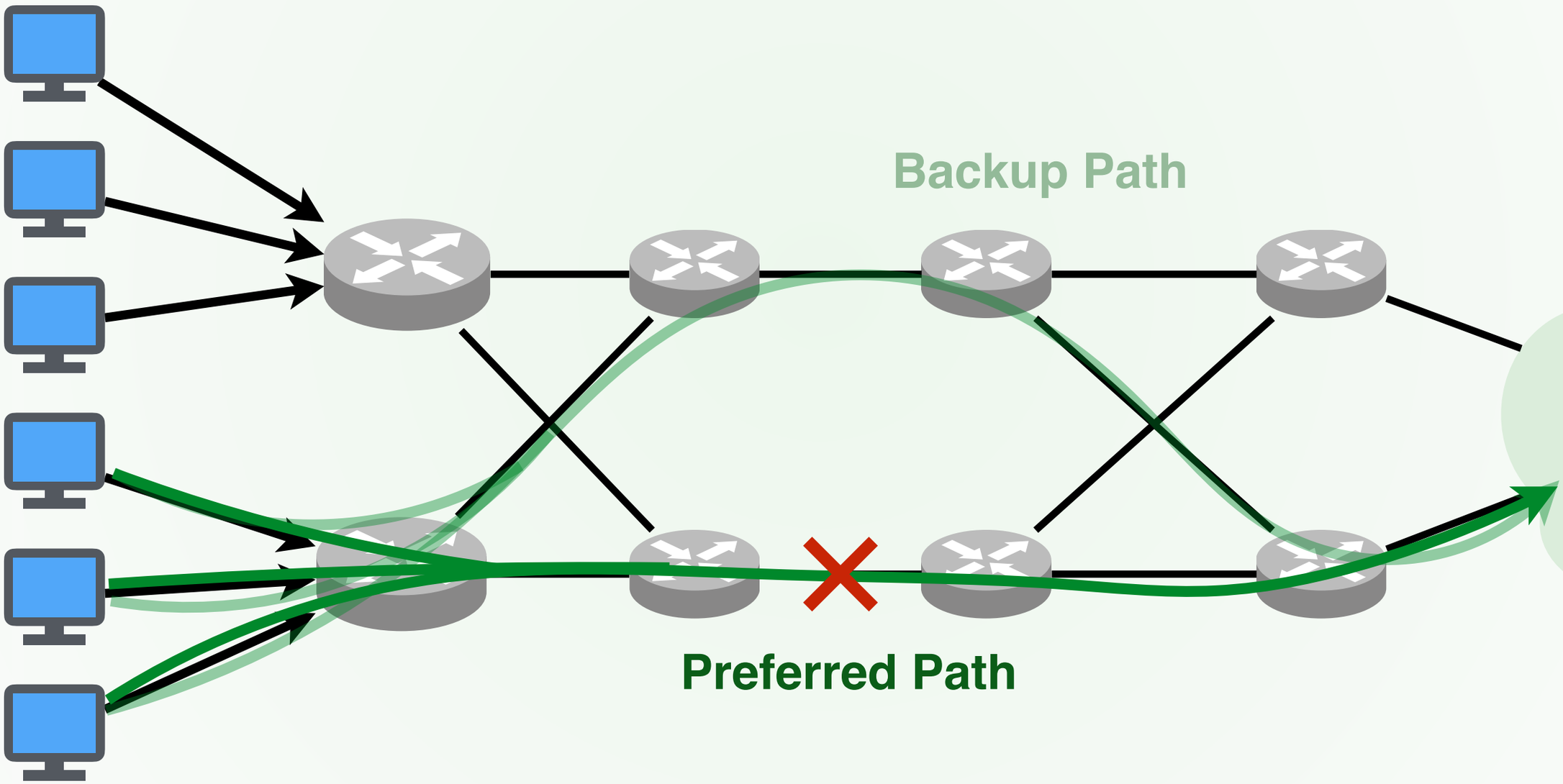
Motivation - Quick Failover



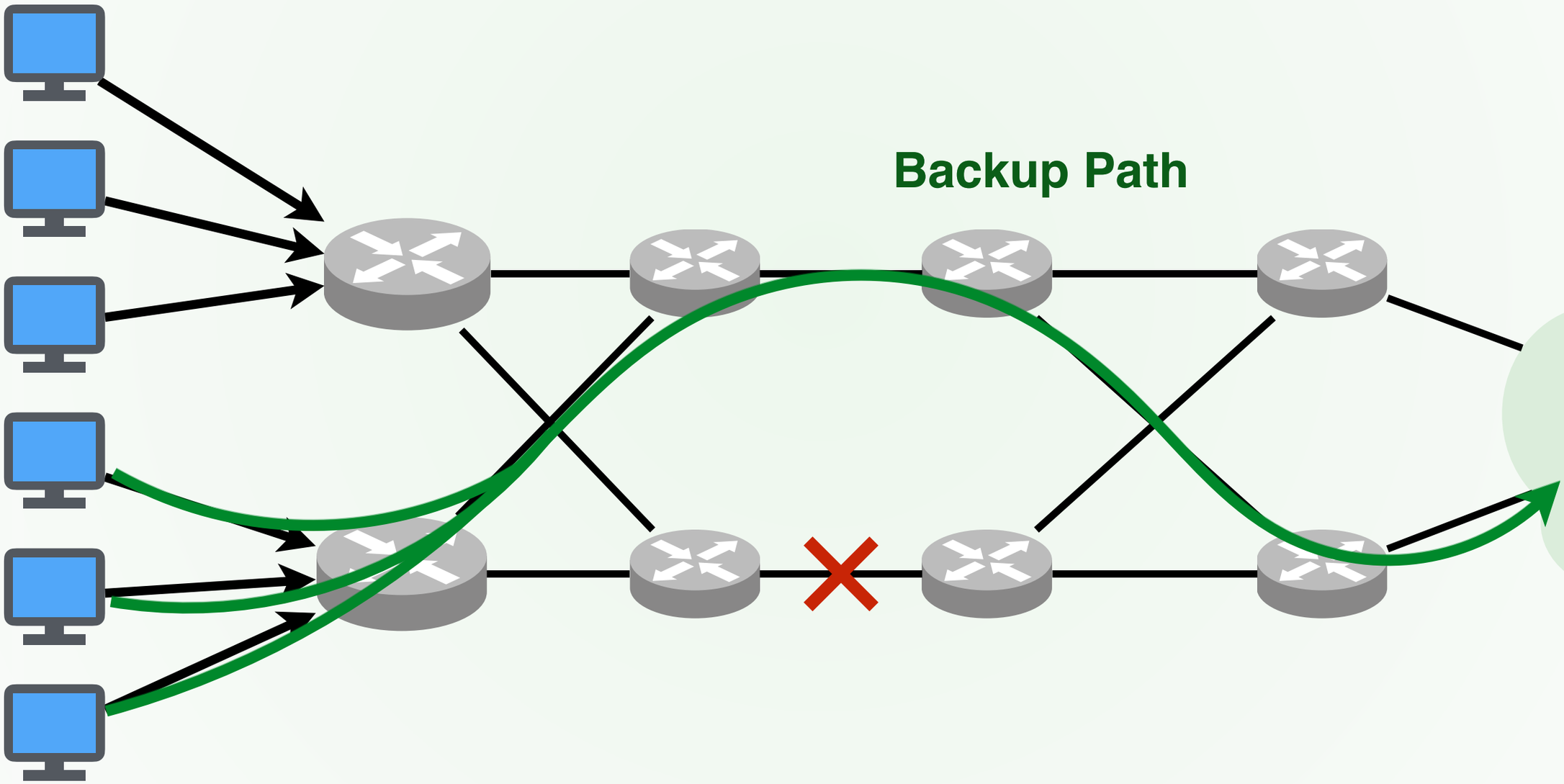
Motivation - Quick Failover



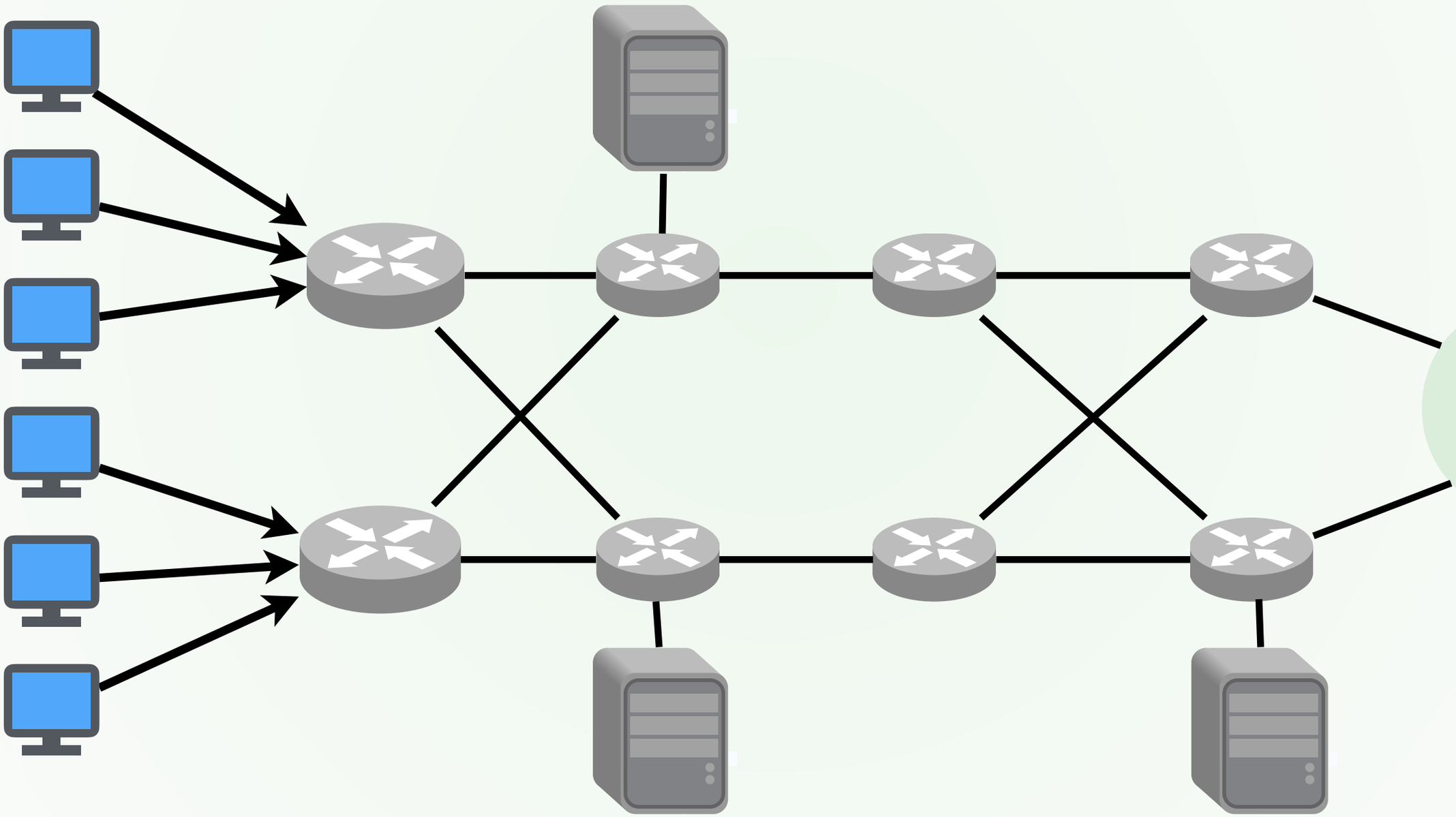
Motivation - Quick Failover



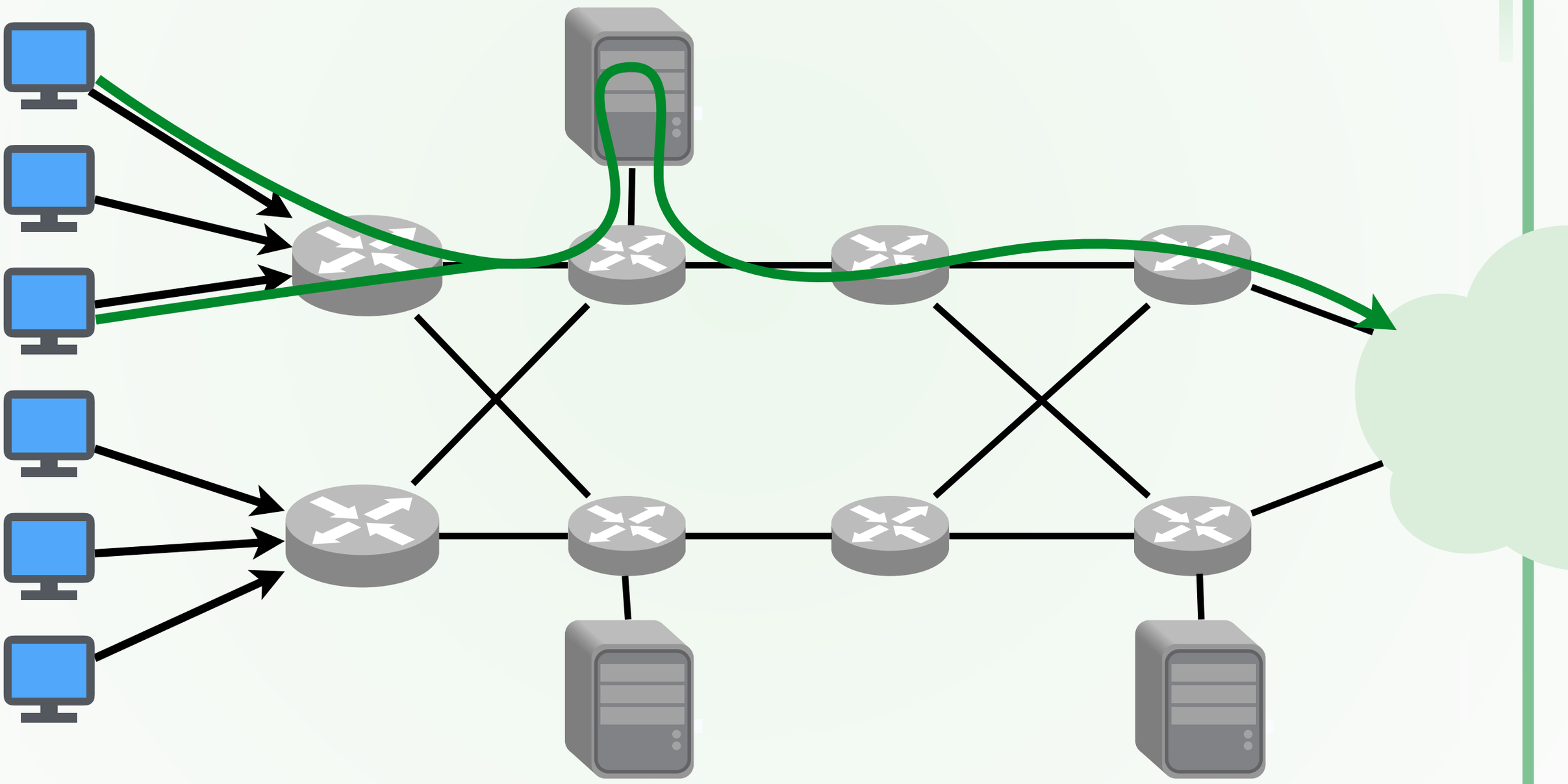
Motivation - Quick Failover



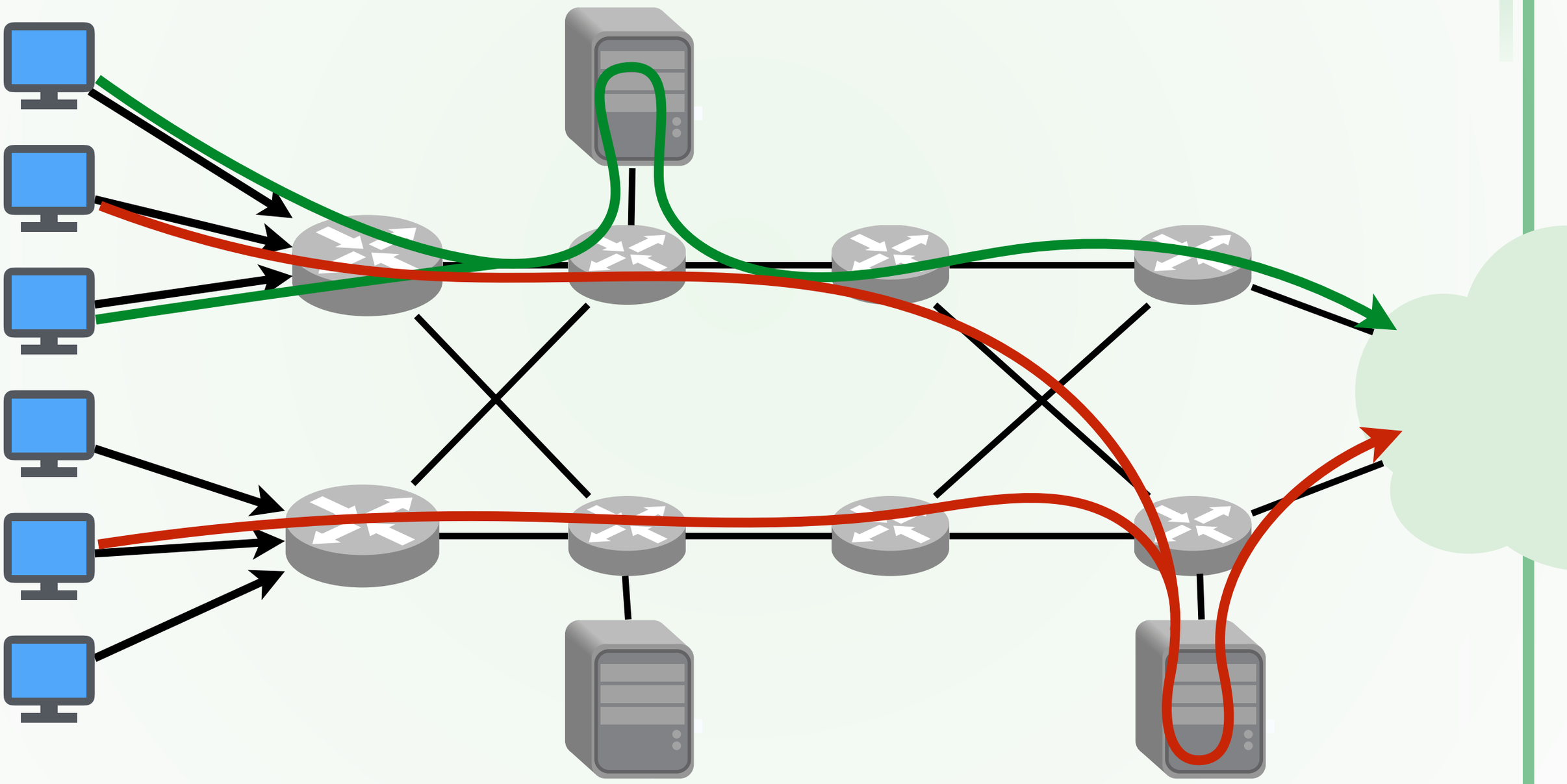
Motivation - Service Chaining



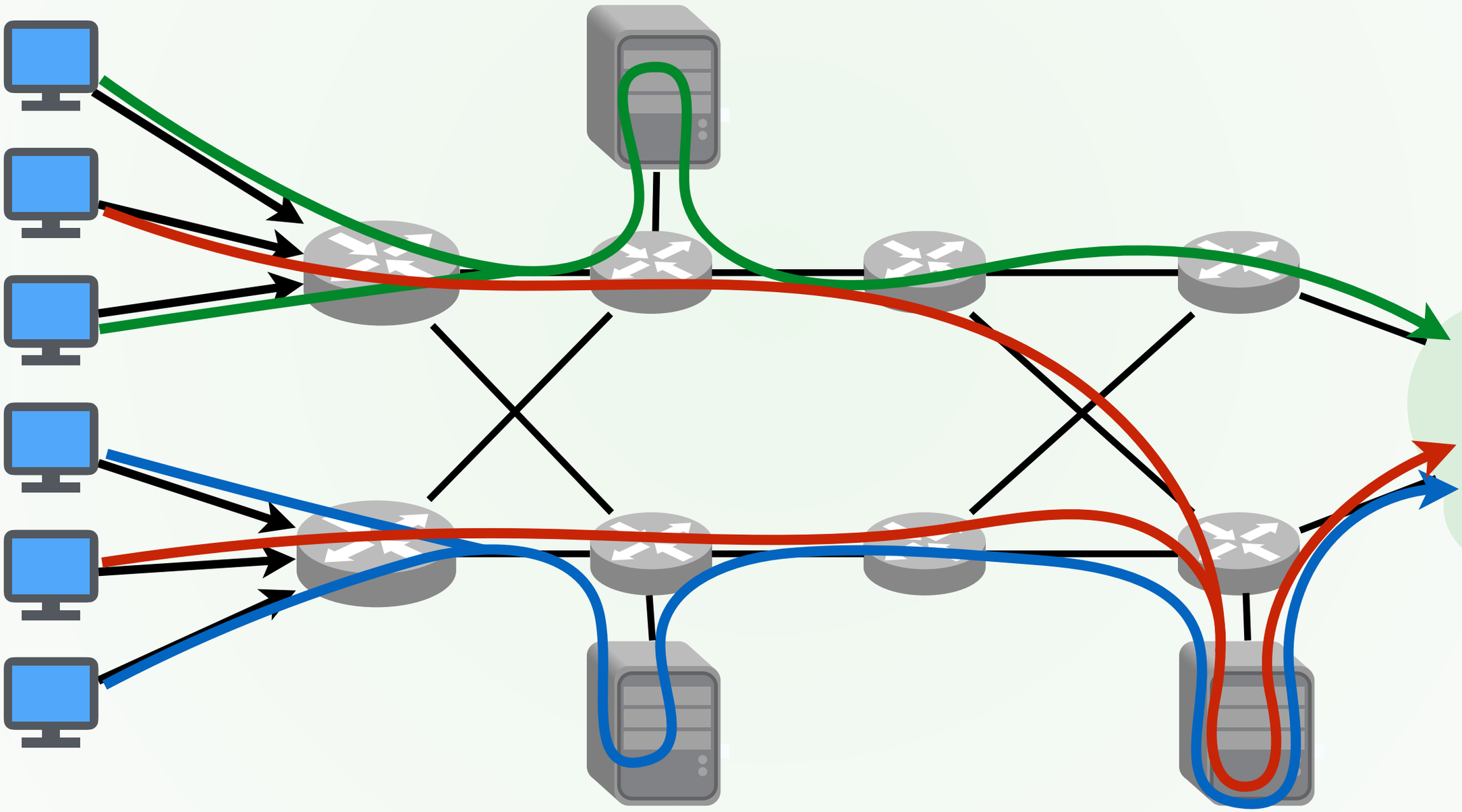
Motivation - Service Chaining



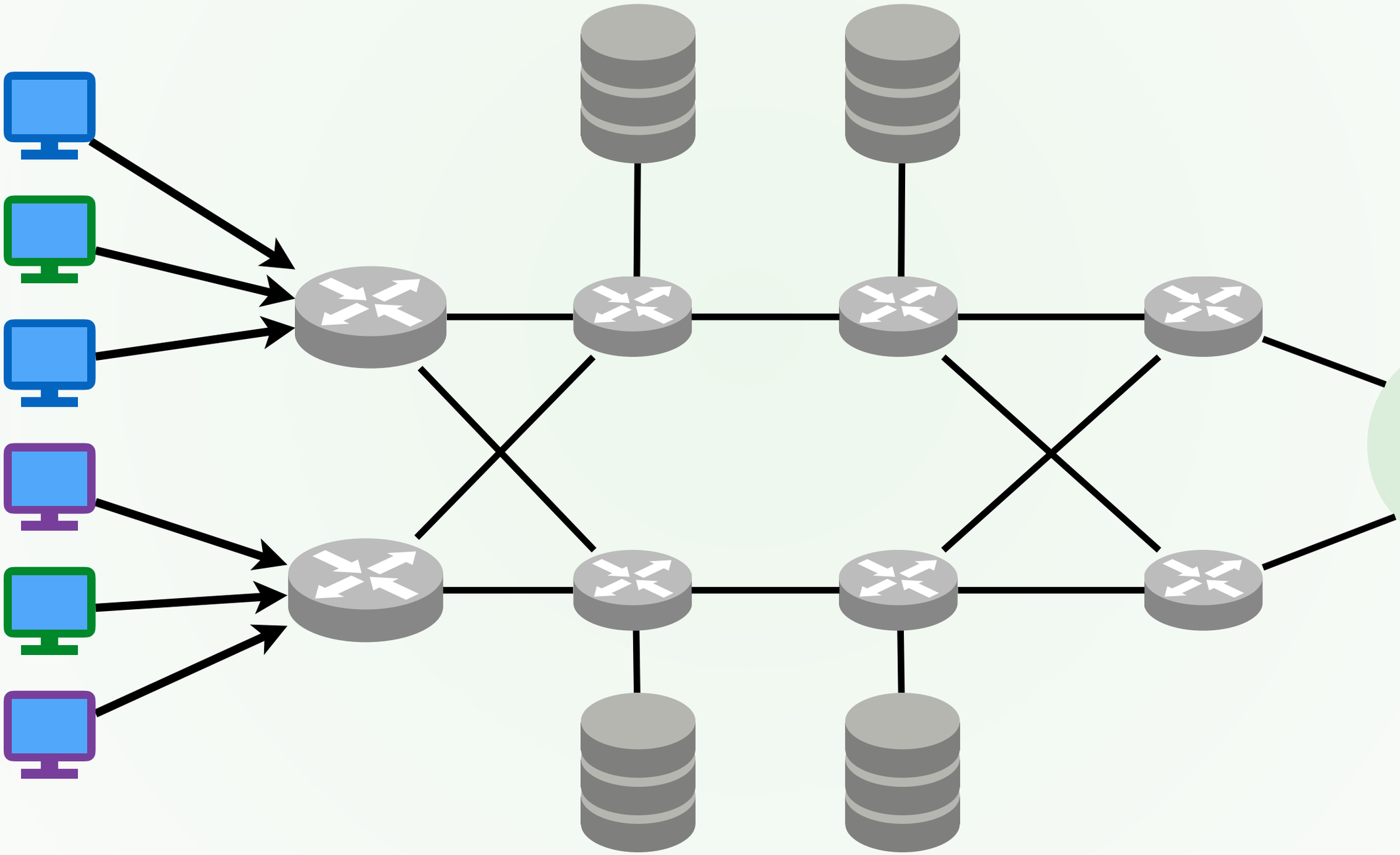
Motivation - Service Chaining



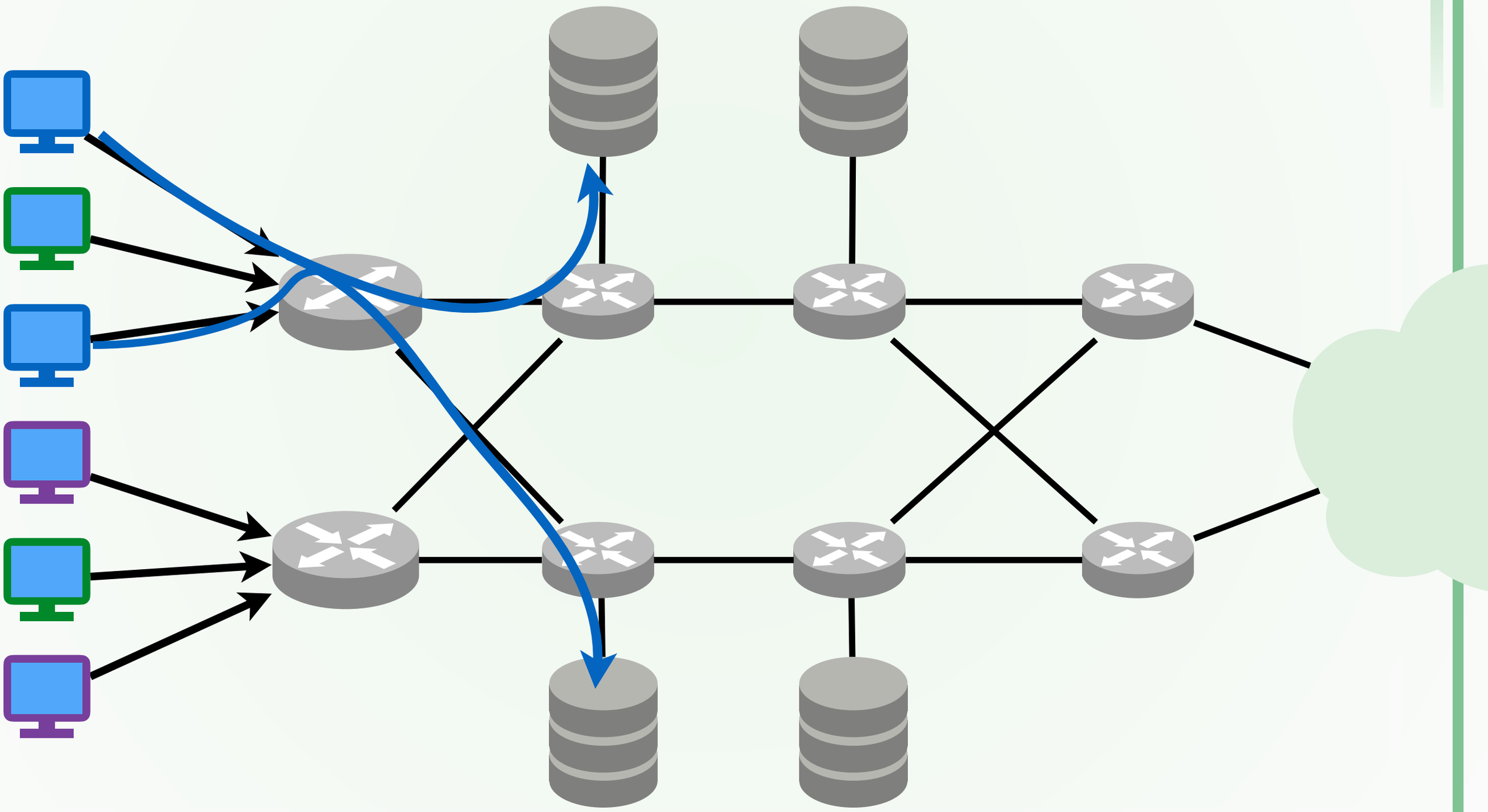
Motivation - Service Chaining



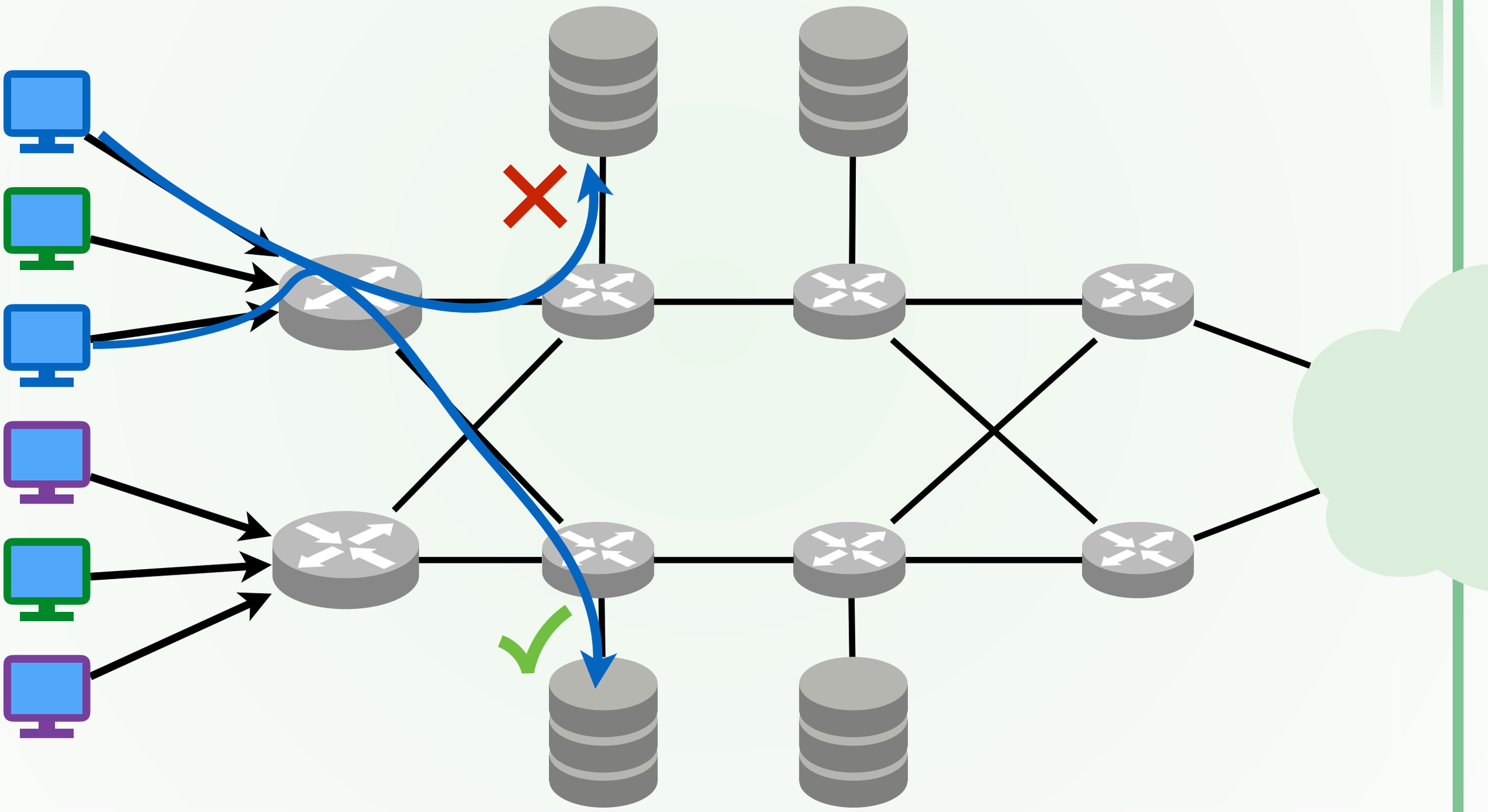
Motivation - Access Control



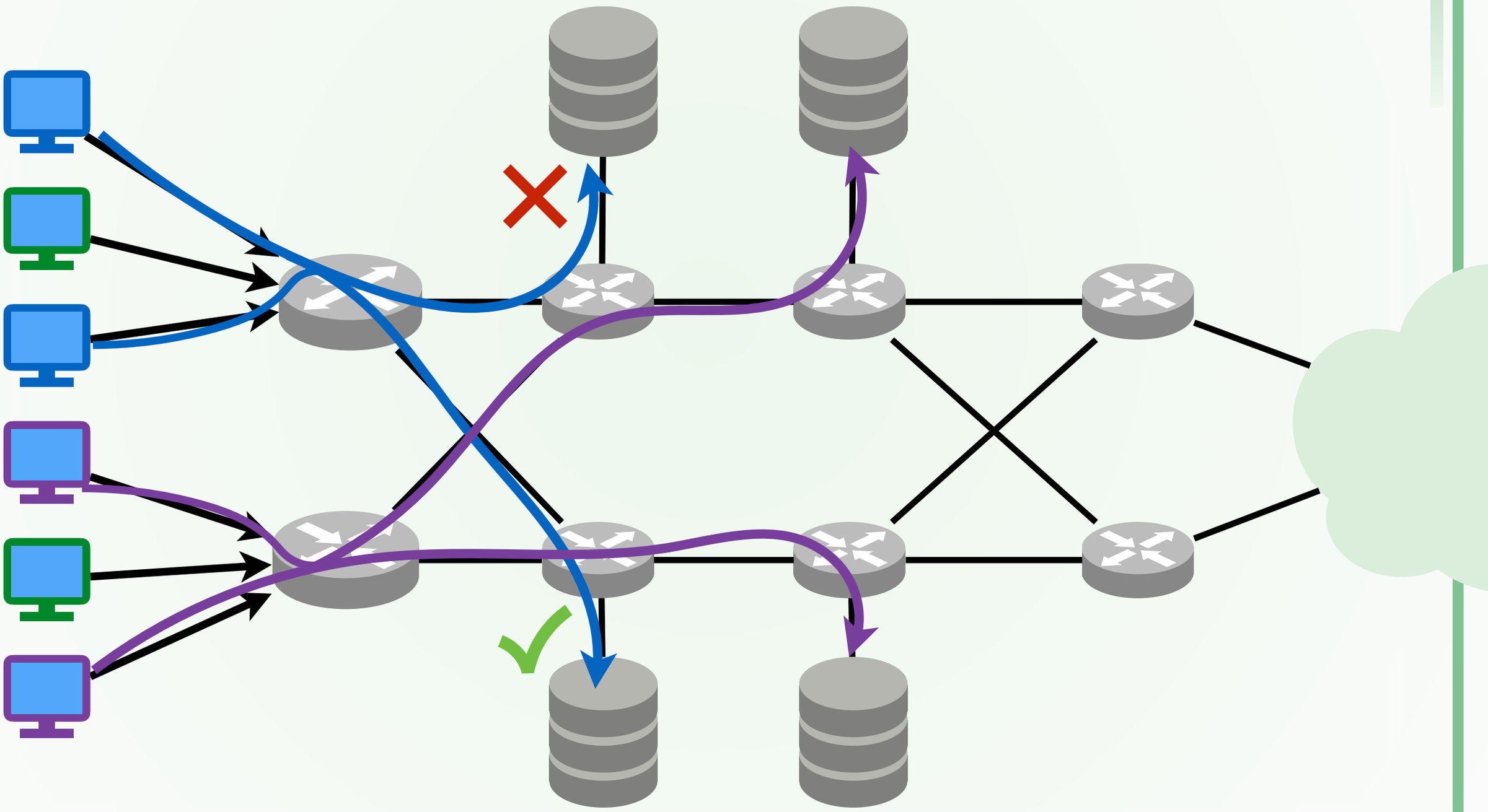
Motivation - Access Control



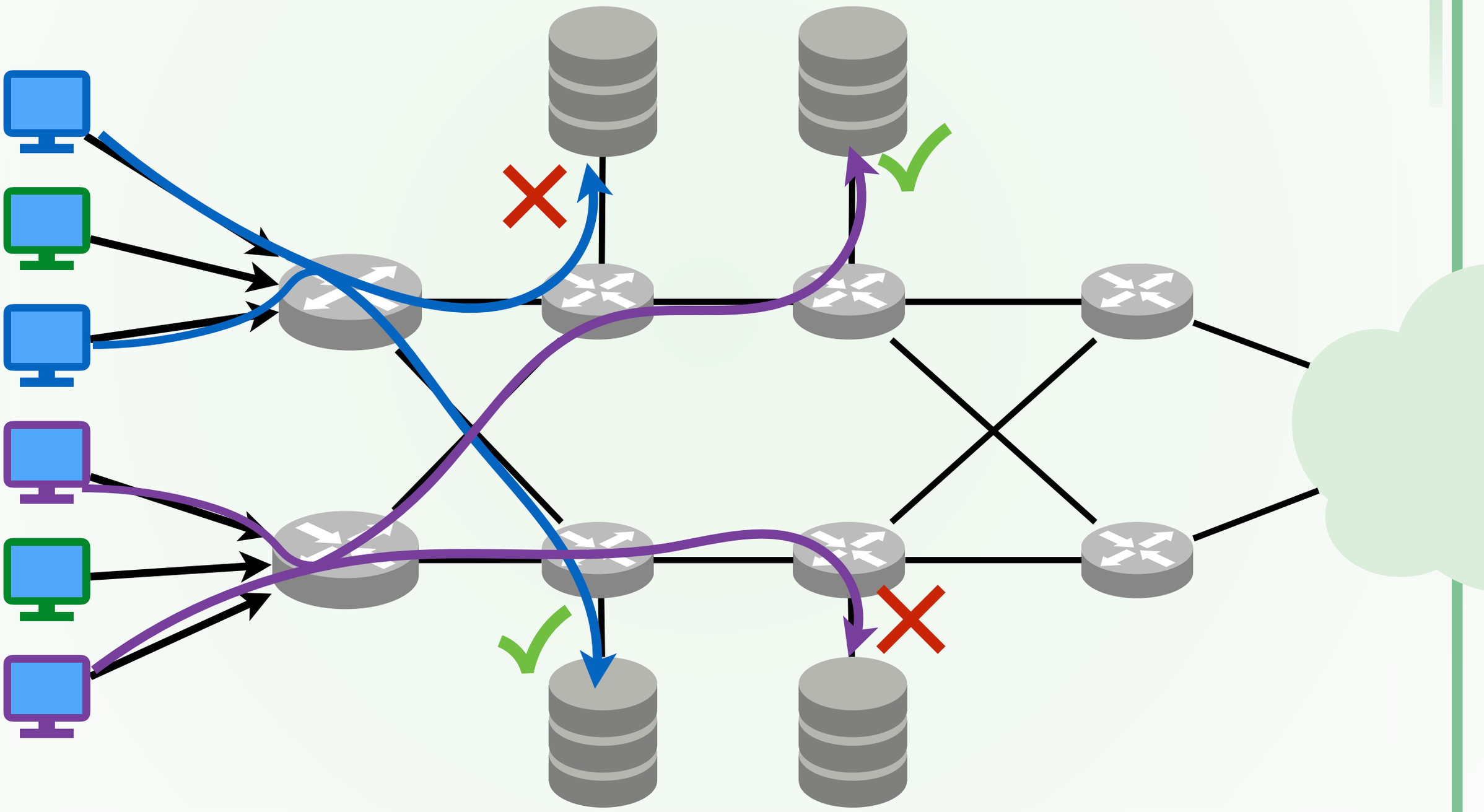
Motivation - Access Control



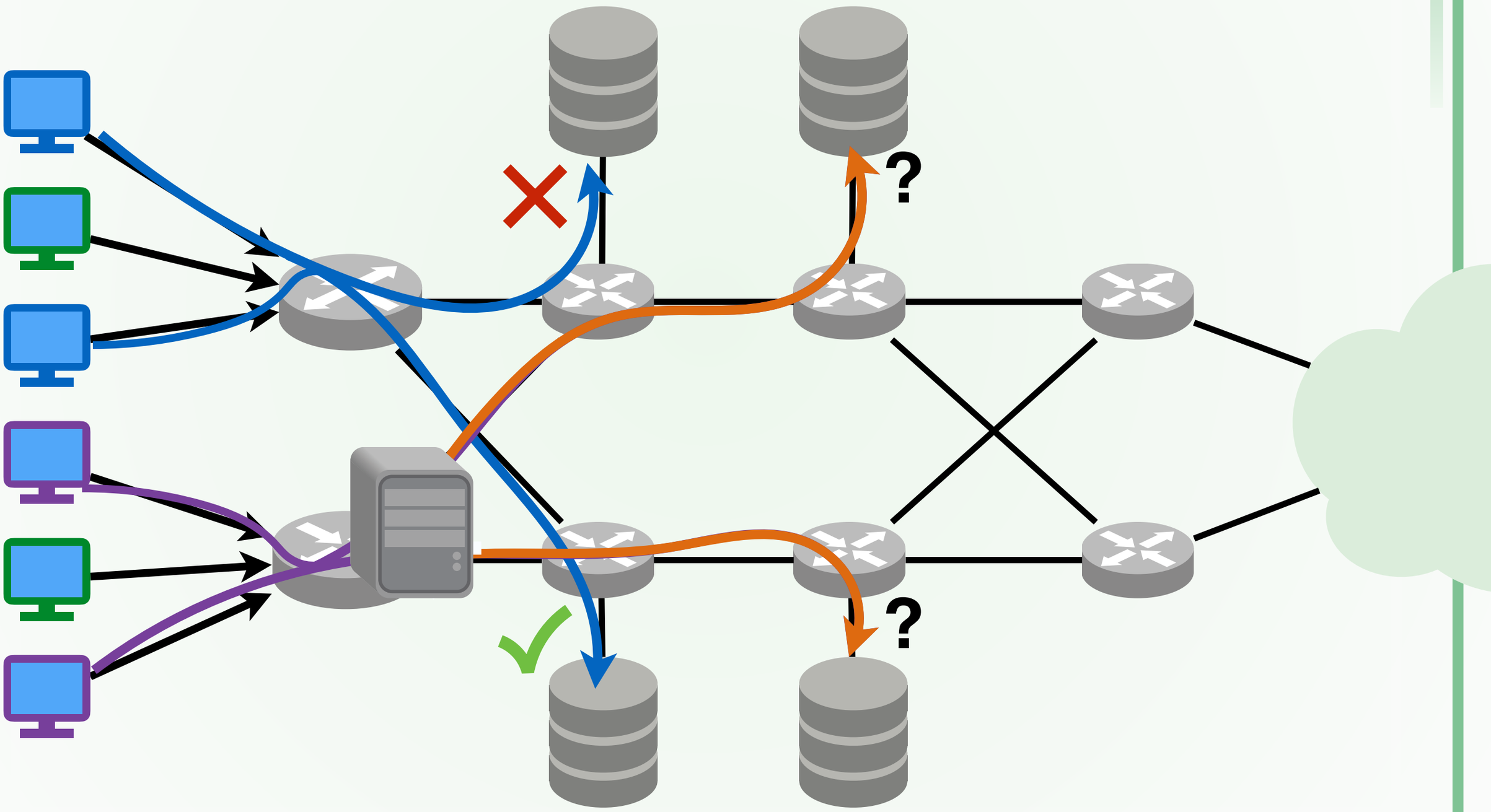
Motivation - Access Control



Motivation - Access Control



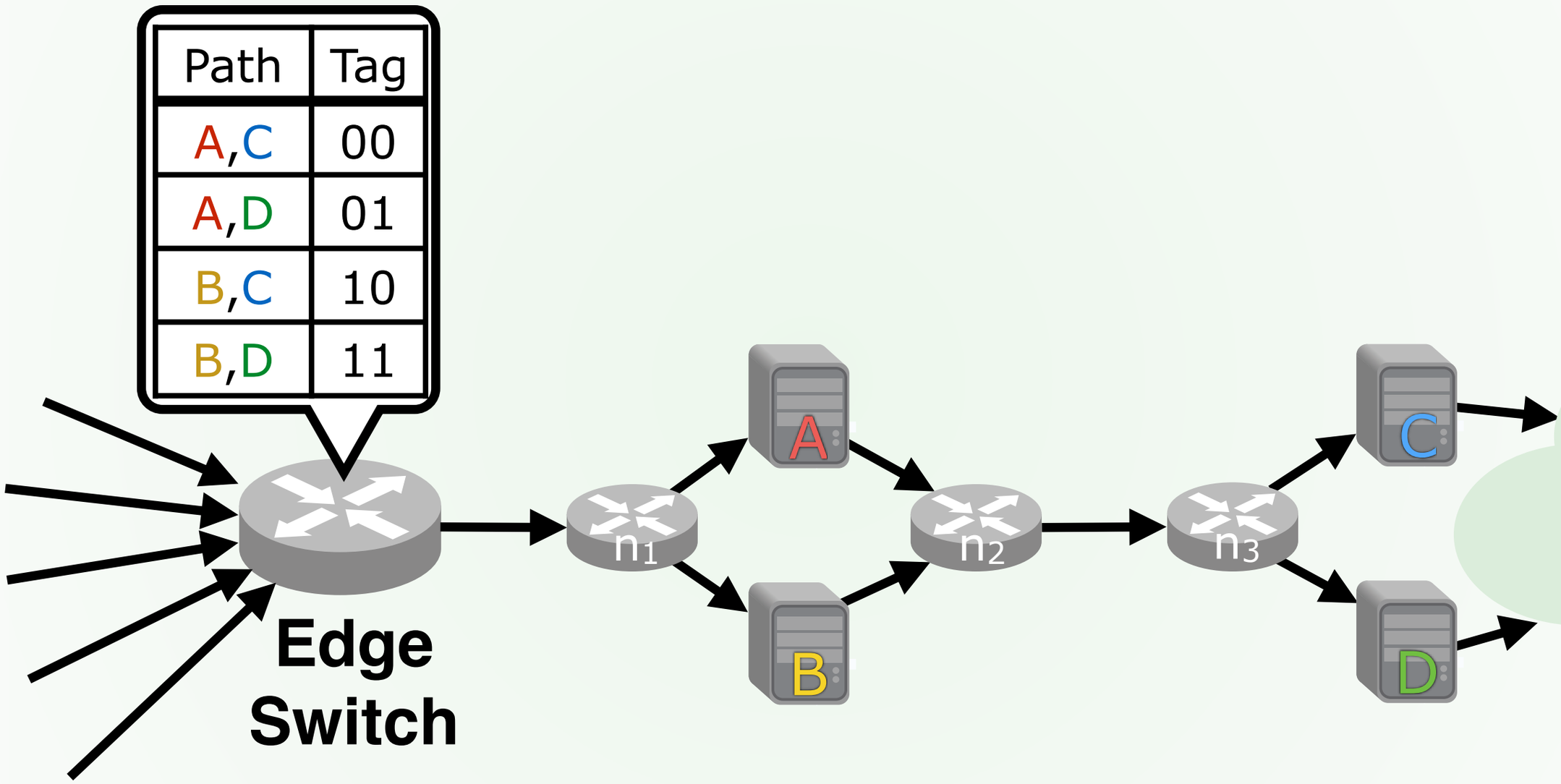
Motivation - Access Control



Tagging Applications

Application	Existing Solution	Tag Field	Tag Conveyed By
Service Chaining	FlowTags	IP Fragment Field	First Middlebox
Policy Enforcement	Alpaca	IP Source Address	DHCP
SDN-Enabled IXP	iSDX	Destination MAC	ARP

Example: Service Chaining



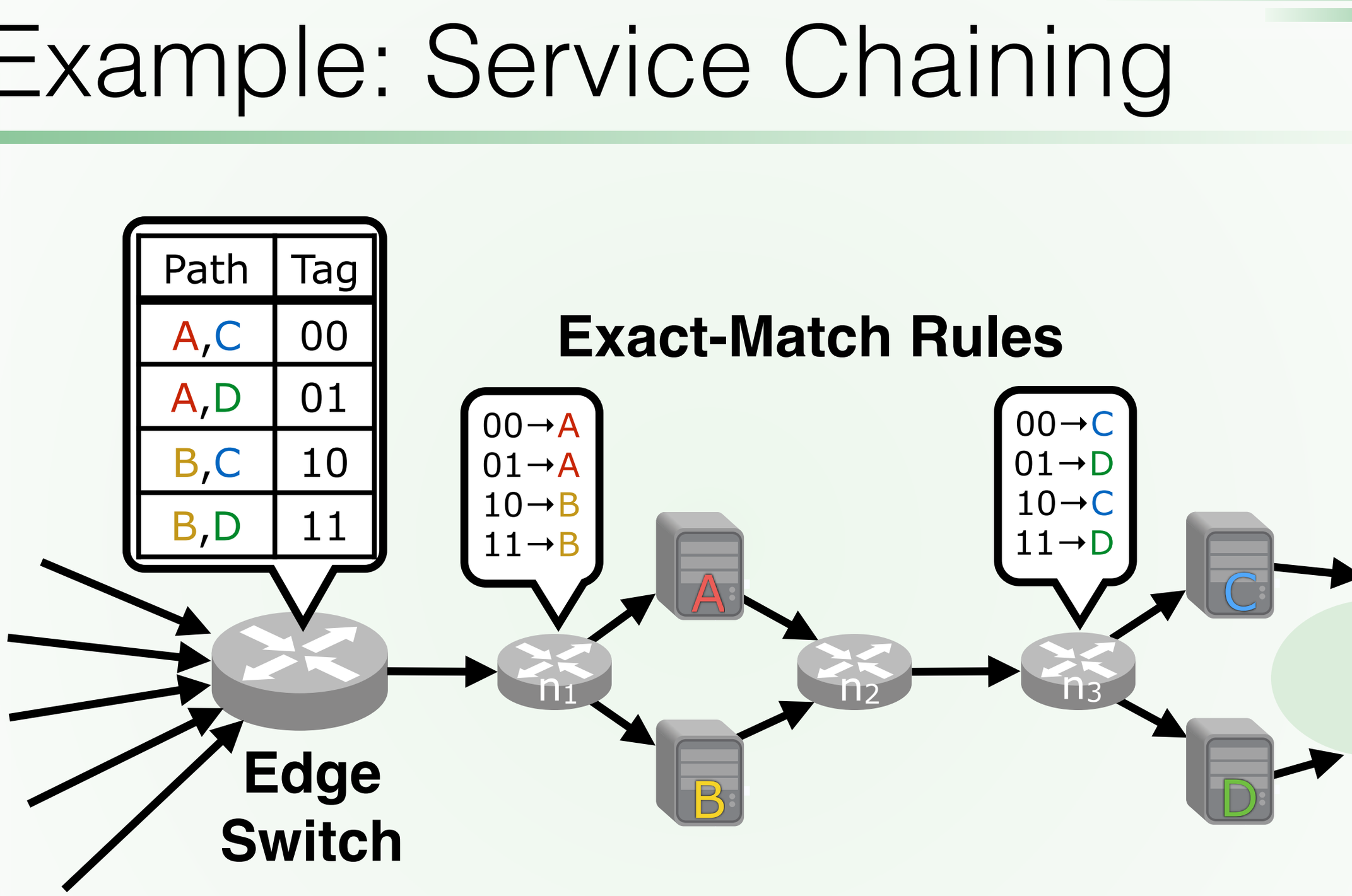
Example: Service Chaining

Path	Tag
A,C	00
A,D	01
B,C	10
B,D	11

Exact-Match Rules

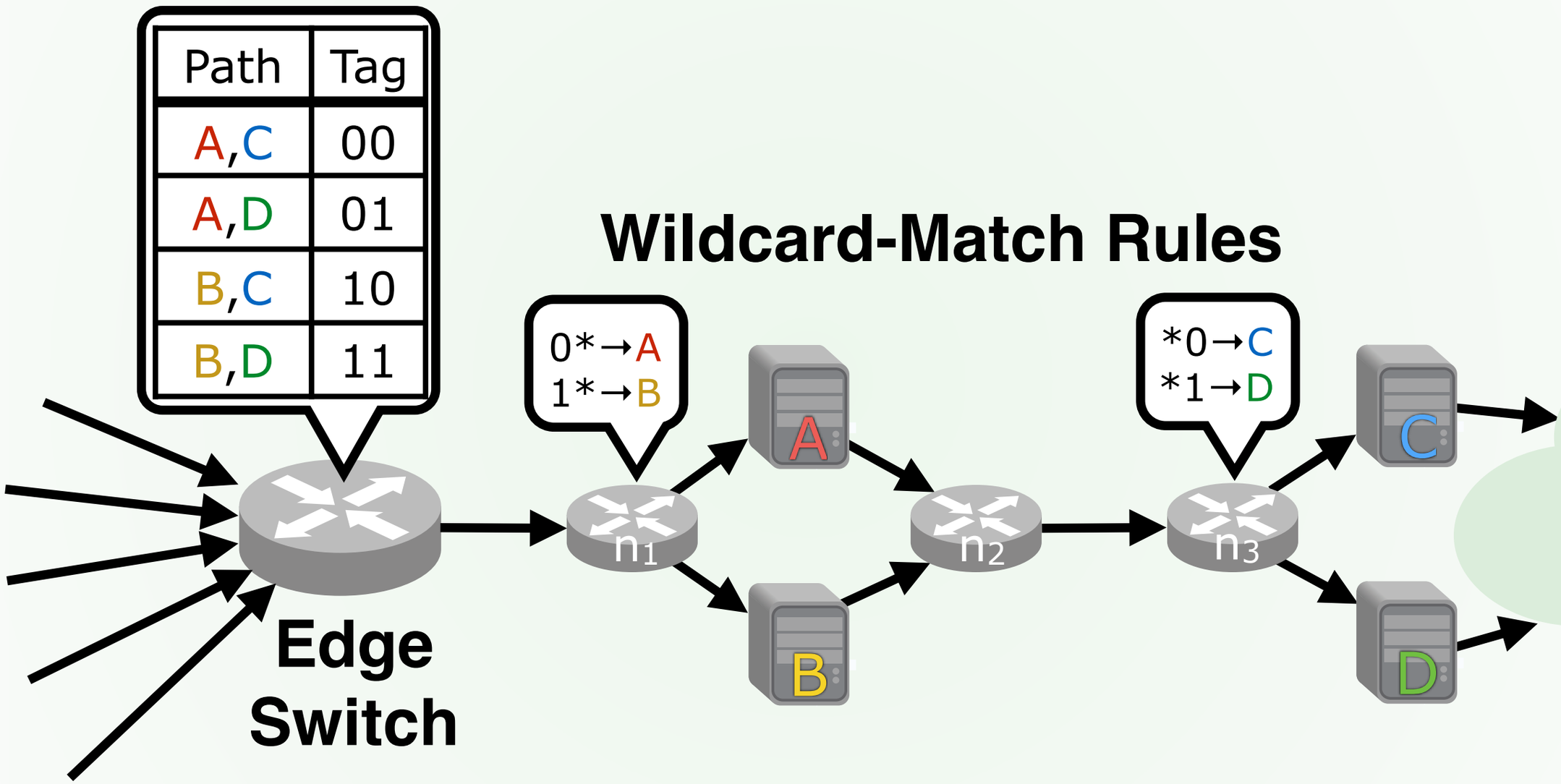
00→A
01→A
10→B
11→B

00→C
01→D
10→C
11→D



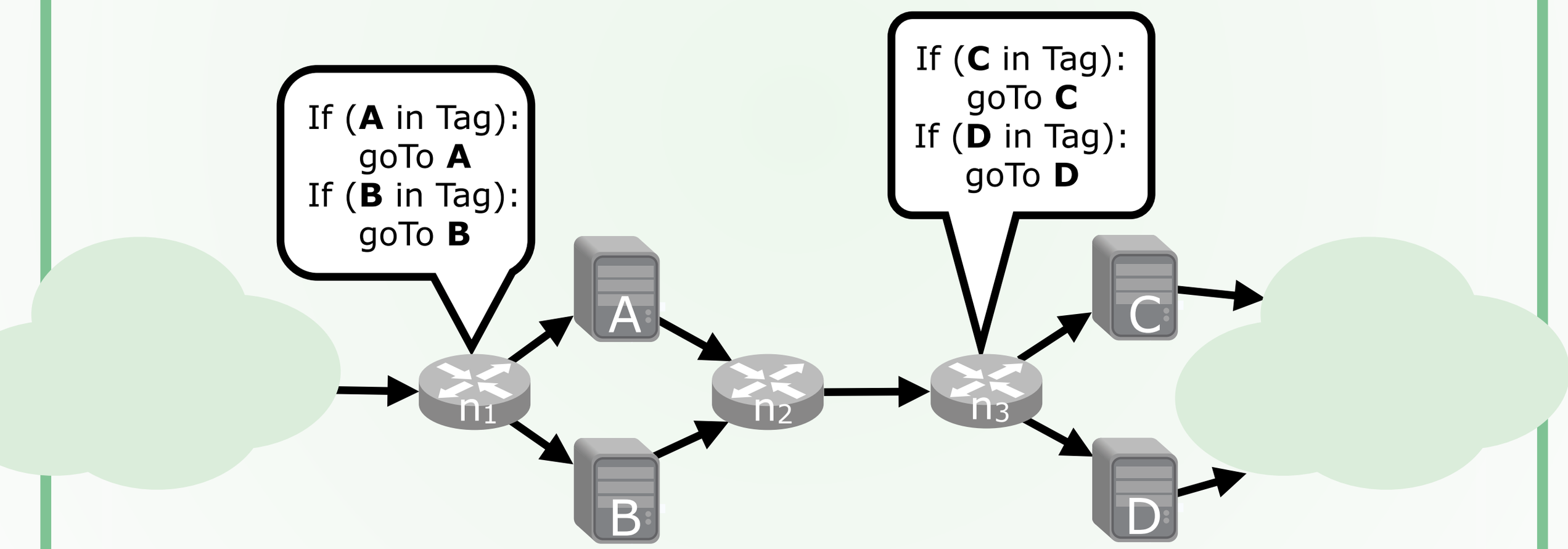
Example: Service Chaining

Path	Tag
A,C	00
A,D	01
B,C	10
B,D	11



Attribute-Encoding Tags

Switch actions often depend on one *attribute*



Tagging Applications

Application	Attributes	Typical Attribute Space Size
Service Chaining	Middleboxes	O(10)
Policy Enforcement	Host Permissions	O(100)
SDN-Enabled IXP	Advertising Peers	O(1000)

Attribute-Encoding Tags

Any tagging problem is composed of two parts:

Attribute-Encoding Tags

Any tagging problem is composed of two parts:

1. A Tag for every FEC

FEC	Attributes	Tag
1	traverse A, traverse C	00
2	hit Mbox A, hit Mbox D	01
3	hit Mbox B, hit Mbox C	10
4	hit Mbox B, hit Mbox D	11

Attribute-Encoding Tags

Any tagging problem is composed of two parts:

1. A Tag for every FEC

FEC	Attributes	Tag
1	traverse A, traverse C	00
2	hit Mbox A, hit Mbox D	01
3	hit Mbox B, hit Mbox C	10
4	hit Mbox B, hit Mbox D	11

2. Pattern-match strings to check for attributes

Attribute	Match Condition
hit Mbox A	Compare Tag to 0*
hit Mbox B	Compare Tag to 1*
hit Mbox C	Compare Tag to *0
hit Mbox D	Compare Tag to *1

Attribute-Encoding Tags

Any tagging problem is composed of two parts:

1. A Tag for every FEC

FEC	Attributes	Tag
1	traverse A, traverse C	00
2	hit Mbox A, hit Mbox D	01
3	hit Mbox B, hit Mbox C	10
4	hit Mbox B, hit Mbox D	11

2. Pattern-match strings to check for attributes

Attribute	Match Condition
hit Mbox A	Compare Tag to 0*
hit Mbox B	Compare Tag to 1*
hit Mbox C	Compare Tag to *0
hit Mbox D	Compare Tag to *1

Tradeoff: Tag width vs. complexity of match conditions

PathSets Outline

1. **Construct tagging scheme for unordered sets of attributes**
2. Extend scheme to support ordered sequences of attributes
3. Using prefix codes to reduce tag size

Strawman Approach

Attribute Sets

FEC	Attributes
S_1	B, C
S_2	B, C, D
S_3	D
S_4	D, E
S_5	D, E, F

Attribute Vectors

FEC	Attributes
S_1	B C _ _ _
S_2	B C D _ _
S_3	_ _ D _ _
S_4	_ _ D E _
S_5	_ _ D E F

Strawman Approach

Attribute Vectors

FEC	Attributes
S_1	B C _ _ _
S_2	B C D _ _
S_3	_ _ D _ _
S_4	_ _ D E _
S_5	_ _ D E F

Masks over [B,C,D,E,F] →

Vector Bitmasks

FEC	Bitmask
S_1	11000
S_2	11100
S_3	00100
S_4	00110
S_5	00111

Strawman Approach

Very simple match rules!

Tags

Set	Bitmask
B,C	11000
B,C,D	11100
D	00100
D,E	00110
D,E,F	00111

Match Patterns

Attribute	Match
B	1****
C	*1***
D	**1**
E	***1*
F	****1

Strawman Approach

Problem: Tag size is linear in the number of attributes to encode. Scales poorly

Set	Bitmask
B,C	11000
B,C,D	11100
D	00100
D,E	00110
D,E,F	00111

Masking over Clusters

	Attributes
S_1	B C _ _ _
S_2	B C D _ _
S_3	_ _ D _ _
S_4	_ _ D E _
S_5	_ _ D E F

Subsets of [B,C,D,E,F] →

FEC	Bitmask
S_1	11000
S_2	11100
S_3	00100
S_4	00110
S_5	00111

Masking over Clusters

	Attributes
S_1	B C _ _ _
S_2	B C D _ _
S_3	_ _ D _ _
S_4	_ _ D E _
S_5	_ _ D E F

Subsets of [B,C,D,E,F] →

FEC	Bitmask
S_1	11000
S_2	11100
S_3	00100
S_4	00110
S_5	00111

Masking over Clusters

	Attributes
S_1	B C _ _ _
S_2	B C D _ _
S_3	_ _ D _ _
S_4	_ _ D E _
S_5	_ _ D E F

Subsets of [B,C,D,E,F]

FEC	Bitmask
S_1	11000
S_2	11100
S_3	00100
S_4	00110
S_5	00111

Masking over Clusters

	Attributes
S_1	B C _ _ _
S_2	B C D _ _
S_3	_ _ D _ _
S_4	_ _ D E _
S_5	_ _ D E F

Subsets of Cluster [B,C,D]

Subsets of Cluster [D,E,F]

	BCD	DEF
S_1	110	
S_2	111	
S_3	001	100
S_4		110
S_5		111

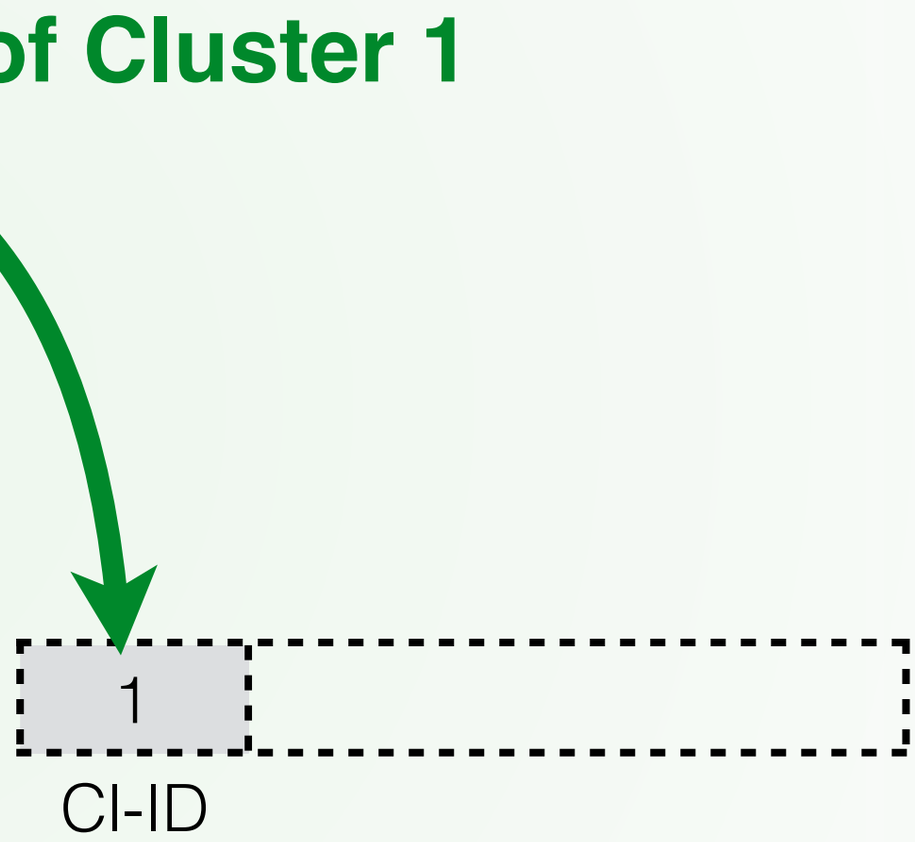
Two-part Tag

Cluster-0 Cluster-1

	BCD	DEF
S_1	110	
S_2	111	
S_3	001	100
S_4		110
S_5		111

Two-part Tag

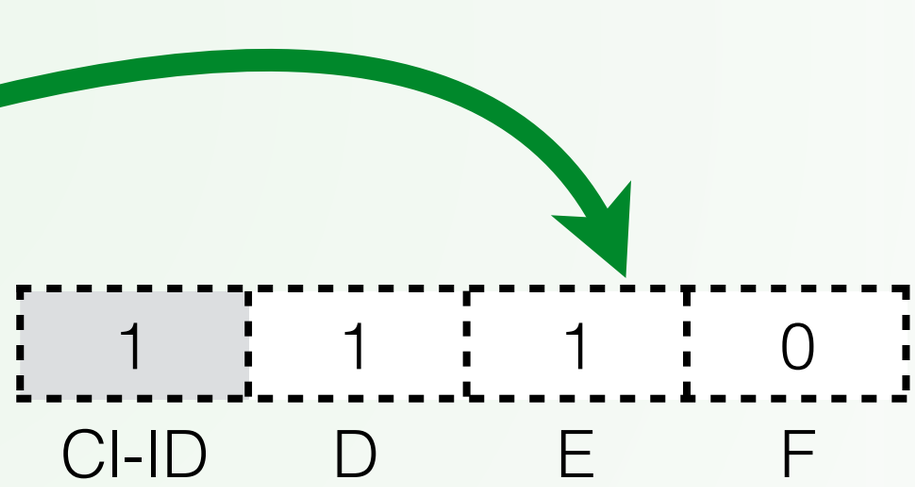
	BCD	DEF
S_1	110	
S_2	111	
S_3	001	100
S_4		110
S_5		111



Two-part Tag

	BCD	DEF
S_1	110	
S_2	111	
S_3	001	100
S_4		110
S_5		111

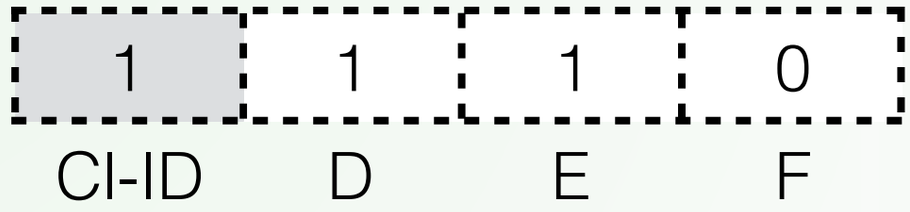
Mask of Cluster 1



Two-part Tag

	BCD	DEF
S_1	110	
S_2	111	
S_3	001	100
S_4		110
S_5		111

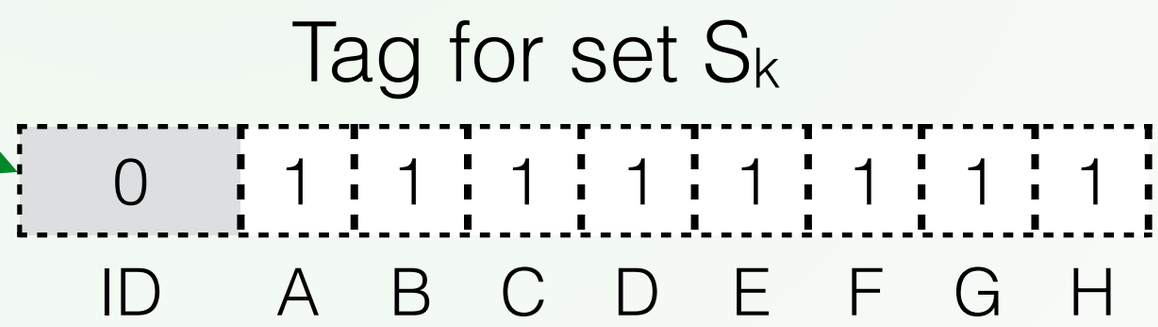
- 4 bits instead of strawman's 5
- Tag Size now = $\log_2$(Num Clusters) + Cluster Size



Min Mask Size

- Tag field at least as big as the largest set
- Ok if assume sets are sparse

	Attributes
.	.
.	.
S_k	ABCDEFGH
.	.
.	.



Matching not as easy

- If X appears in multiple clusters, then multiple match patterns needed for X

	BCD	DEF
S1	110	
S2	111	
S3	001	100
S4		110
S5		111

Matching not as easy

- If X appears in multiple clusters, then multiple match patterns needed for X

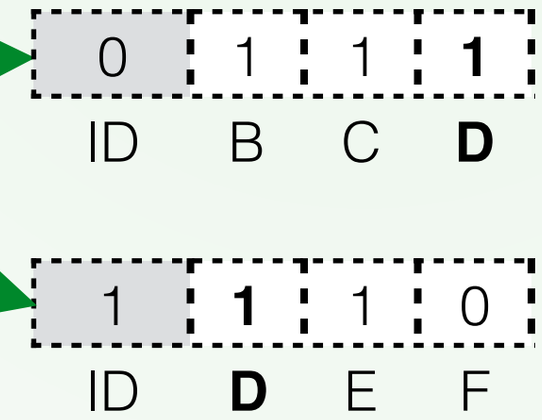
D in both clusters

	BCD	DEF
S1	110	
S2	111	
S3	001	100
S4		110
S5		111

Matching not as easy

- If X appears in multiple clusters, then multiple match patterns needed for X

	BCD	DEF
S1	110	
S2	111	
S3	001	100
S4		110
S5		111



Matching not as easy

- If X appears in multiple clusters, then multiple match patterns needed for X

	BCD	DEF
S1	110	
S2	111	
S3	001	100
S4		110
S5		111

ID	B	C	**D**
0	1	1	**1**

ID	**D**	E	F
1	**1**	1	0

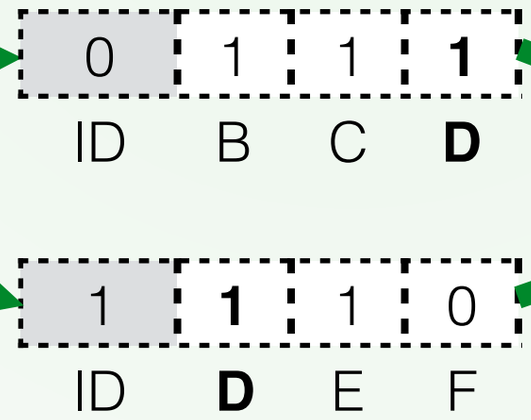
Att	Match
B	01**
C	0*1*
D	**0**1 OR 11****
E	1*1*
F	1**1

Matching not as easy

- If X appears in multiple clusters, then multiple match patterns needed for X

6 patterns (strawman had 5)

	BCD	DEF
S1	110	
S2	111	
S3	001	100
S4		110
S5		111



Att	Match
B	01**
C	0*1*
D	**0**1 OR 11****
E	1*1*
F	1**1

PathSets Outline

1. Construct tagging scheme for unordered sets of attributes

2. **Extend scheme to support ordered sequences of attributes**

3. Using prefix codes to reduce tag size

Ordered Attribute Checks

Sequence
A→B→C→D
C→D→E
B→E
A→C→E

Ordered Attribute Checks

One Cluster - ABCDE - No ID

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Ordered Attribute Checks

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Ordered Attribute Checks

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Att.	Match String
A	"1****"
B	"*1***"
C	"**1**"
D	"***1*"
E	"****1"

Ordered Attribute Checks

Doesn't enforce attribute ordering

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Att.	Match String
A	"1****"
B	"*1***"
C	"**1**"
D	"***1*"
E	"****1"

Ordered Attribute Checks

Sequences ordered **Left-to-Right**

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Att.	Match String
A	"1****"
B	"*1***"
C	"**1**"
D	"***1*"
E	"****1"

Ordered Attribute Checks

Sequences ordered **Left-to-Right**

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Att.	Match String
A	"1****"
B	"**0**1***"
C	"**00**1**"
D	"**000**1*"
E	"**0000**1"

Leftmost attribute takes priority

Attribute Sequences

If all sequences adhere to some *supersequence*, the extension is straightforward

Supersequence	ID
A→B→C→D→E	-

Sequence	Tag
A→B→C→D	11110
C→D→E	00111
B→E	01001
A→C→E	10101

Att.	Match String
A	“1****”
B	“01***”
C	“001**”
D	“0001*”
E	“00001”

Real Applications not ideal

Attribute Sequence
A→B →C
B →C→D
A→C→B
C→B →D

B comes before C

B comes after C

Real Applications not ideal

Attribute Sequence
$A \rightarrow B_1 \rightarrow C$
$B_1 \rightarrow C \rightarrow D$
$A \rightarrow C \rightarrow B_2$
$C \rightarrow B_2 \rightarrow D$

B_1 comes before C

B_2 comes after C

Real Applications not ideal

Attribute Sequence
$A \rightarrow B_1 \rightarrow C$
$B_1 \rightarrow C \rightarrow D$
$A \rightarrow C \rightarrow B_2$
$C \rightarrow B_2 \rightarrow D$

B_1 comes before C

B_2 comes after C

Total Order = $A < \mathbf{B_1} < C < \mathbf{B_2} < D$

Real Applications not ideal

Attribute Sequence	Tag
$A \rightarrow B_1 \rightarrow C$	11100
$B_1 \rightarrow C \rightarrow D$	01101
$A \rightarrow C \rightarrow B_2$	10110
$C \rightarrow B_2 \rightarrow D$	00111

Total Order = $A < \mathbf{B_1} < C < \mathbf{B_2} < D$

Real Applications not ideal

Attribute Sequence	Tag
$A \rightarrow B_1 \rightarrow C$	11100
$B_1 \rightarrow C \rightarrow D$	01101
$A \rightarrow C \rightarrow B_2$	10110
$C \rightarrow B_2 \rightarrow D$	00111

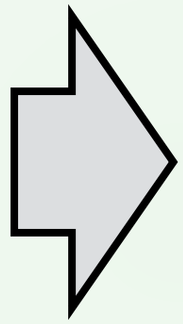
Att	Match Pattern
A	1****
B	**01*** OR 0001***
C	001**
D	00001

Total Order = A < **B_1** < C < **B_2** < D

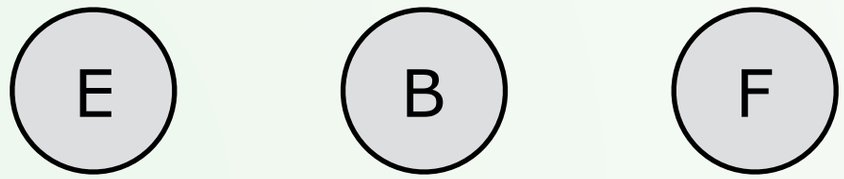
How do we systematically identify and resolve conflicts efficiently?

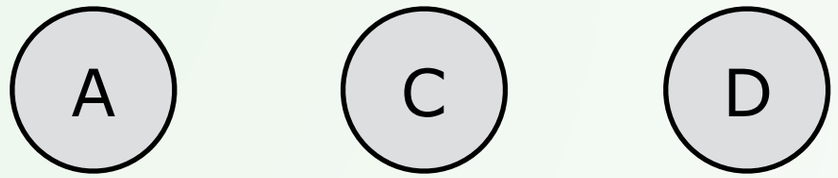
Heuristic Intuition

Sequences
A→C→B→D
E→A→B
D→F
A→B→C→D



Sequence Graph

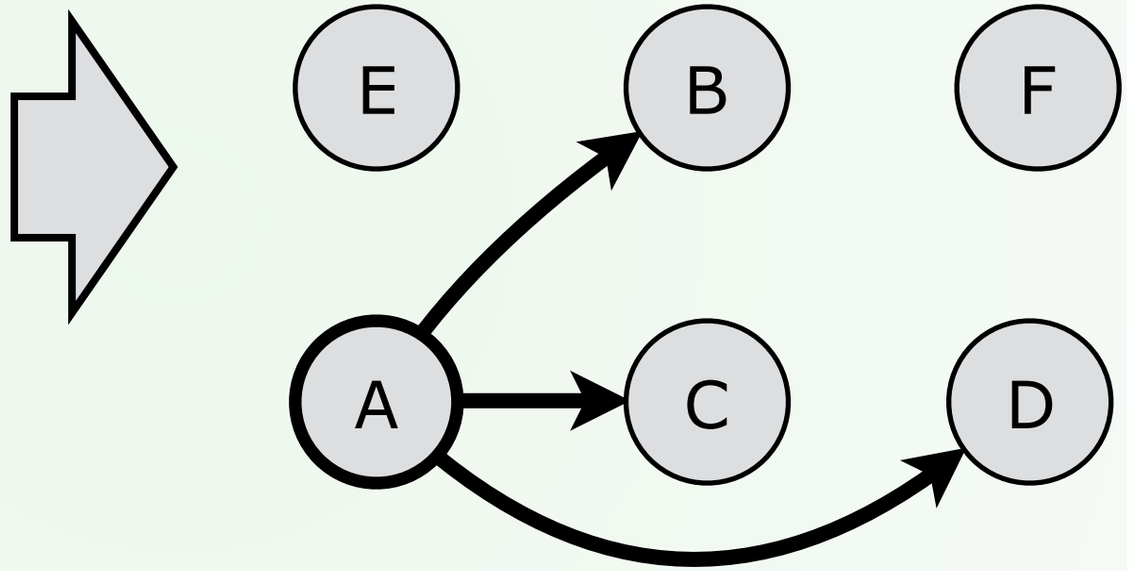




Vertex for each attribute

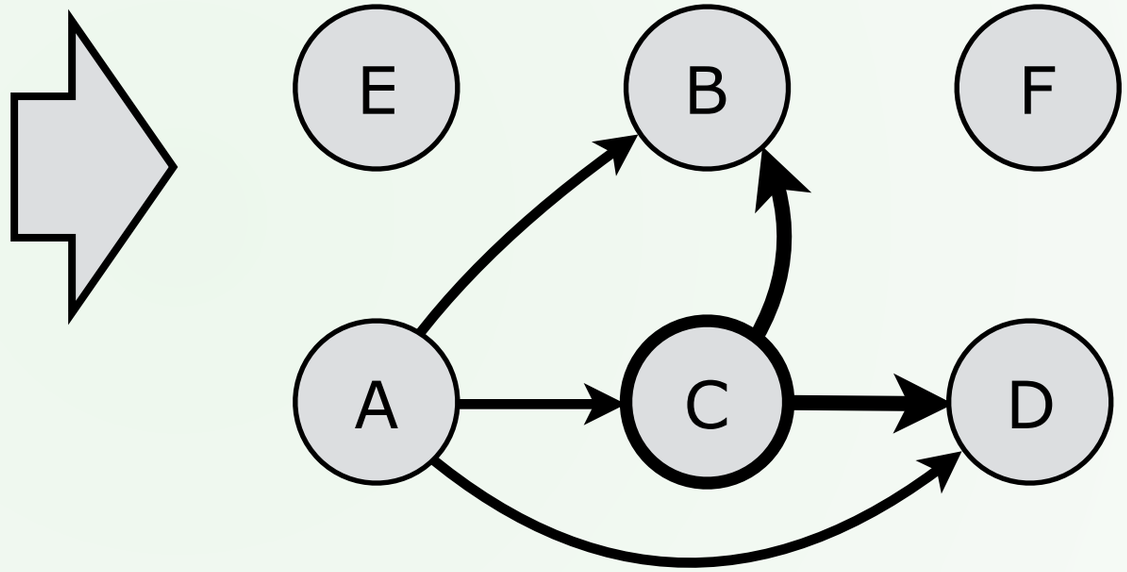
Heuristic Intuition

Sequences
A→C→B→D
E→A→B
D→F
A→B→C→D



Heuristic Intuition

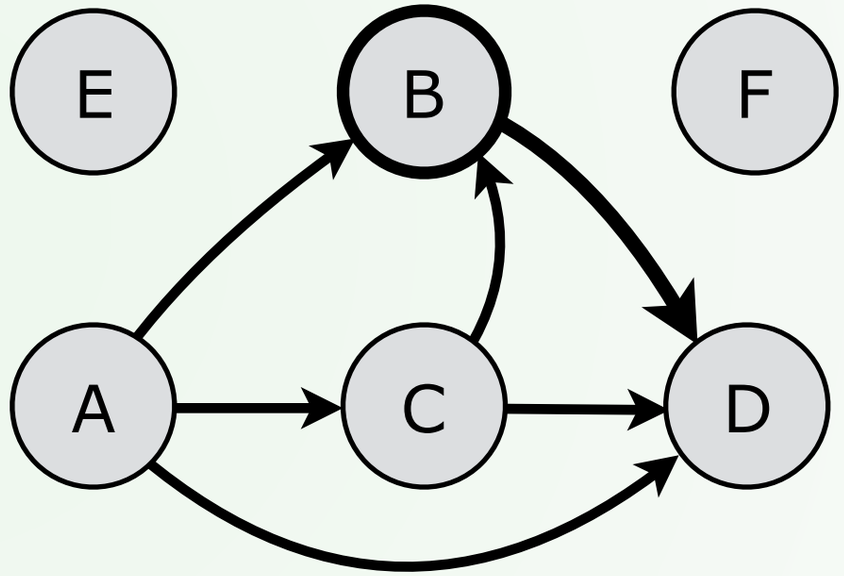
Sequences
A→**C**→B→D
E→A→B
D→F
A→B→C→D



Heuristic Intuition

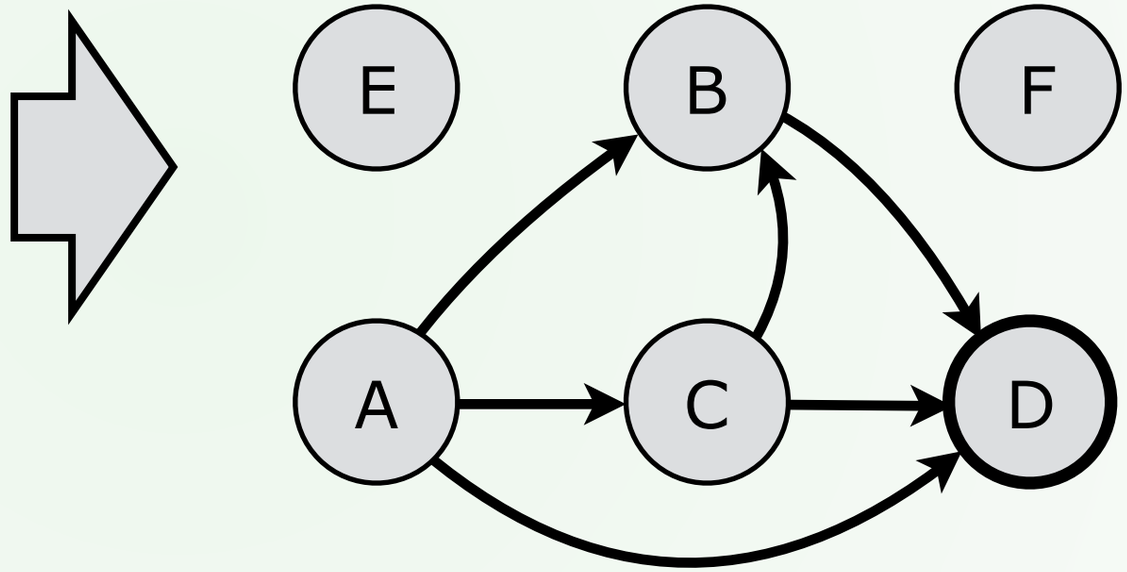
Sequences
A→C→**B**→D
E→A→B
D→F
A→B→C→D

Sequence Graph



Heuristic Intuition

Sequences
A→C→B→**D**
E→A→B
D→F
A→B→C→D

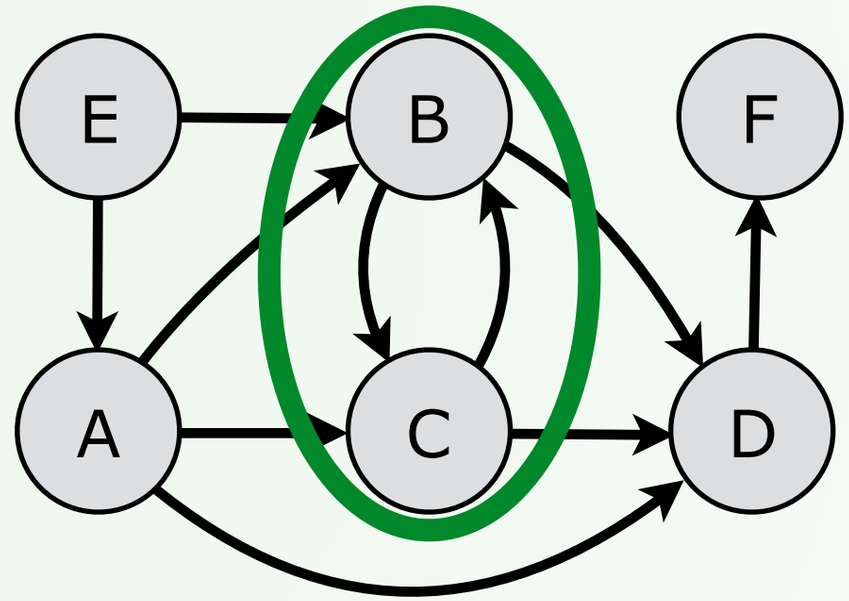


Heuristic Intuition

Sequences
A→C→B→D
E→A→B
D→F
A→B→C→D

Repeat for all sequences

Sequence Graph



Use existing graph theory to identify and resolve conflicts

PathSets Outline

1. Construct tagging scheme for unordered sets of attributes
2. Extend scheme to support ordered sequences of attributes
3. **Using prefix codes to reduce tag size**

Fixed-Length IDs are Inefficient

Fixed-Length IDs are Inefficient

Tags can vary in size

ID	Clusters
00	[A,B,C]
01	[C,D]
10	[E,F]
11	[W,X,Y,Z]

Shortest Tag = 4 Bits
Longest Tag = 6 Bits

Fixed-Length IDs are Inefficient

Tags can vary in size

ID	Clusters
00	[A,B,C]
01	[C,D]
10	[E,F]
11	[W,X,Y,Z]

Shortest Tag = 4 Bits
Longest Tag = 6 Bits

Many identifiers can be left unused

ID	Clusters
000	[E,F]
001	[F,H,I]
010	[L,M]
011	[M,N,O,P]
100	[R,S,T]
101	-
110	-
111	-

Prefix-Free Codes as IDs

ID	Clusters
1	[A,B,C,D]
01	[E,F,G]
001	[H,I]
0001	[J]
0000	-

Prefix-Free Codes as IDs

- Observation: Tags are uniquely decodable iff no ID is a prefix of any other

- Can use this to create variable-length identifiers

ID	Clusters
1	[A,B,C,D]
01	[E,F,G]
001	[H,I]
0001	[J]
0000	-

Prefix-Free Codes as IDs

- Observation: Tags are uniquely decodable iff no ID is a prefix of any other

ID	Clusters
1	[A,B,C,D]
01	[E,F,G]
001	[H,I]
0001	[J]
0000	-

- Can use this to create variable-length identifiers

- Use theory of Kraft's Inequality to optimally build identifiers

Prefix-Free Codes as IDs

- Observation: Tags are uniquely decodable iff no ID is a prefix of any other

- Can use this to create variable-length identifiers

- Use theory of Kraft's Inequality to optimally build identifiers

ID	Clusters
1	[A,B,C,D]
01	[E,F,G]
001	[H,I]
0001	[J]
0000	-

Shortest Tag = 5 Bits
Longest Tag = 5 Bits

Empirical Analysis

- Evaluated Scheme for two Applications:

1. Software-Defined IXP Case (Unordered Sets)
2. Middlebox paths (Ordered Sets)

SDN-IXP Evaluation

- Used AMS-IX RIPE RIB dumps (633k prefixes, 63 attached networks)
- Generated 1k random SDN policies, computed switch table size

Statistics

- Balancing tradeoffs, tags that required 37 bits used <2k switch entries
- Flat tagging requires ~18 bits, >200k entries

Service Chaining Evaluation

- 800 random paths using Markov chains over some hidden super sequence
- 5% chance of pairwise reordering
- 40 distinct middlebox types

Statistics

- Flat tagging needs 10 bits, ~150 entries per switch
- PathSets needs <19 bits, ~75 entries

Conclusions

- Our tagging scheme is general enough to be used by many applications

- A first step for non-flat attribute-encoding tagging

- Prototype code publicly available:
  github.com/PrincetonUniversity/PathSets

Thank You!