Applying the Migration-based Approach to Database Delivery

Vladimir Khorikov
PROGRAMMER

@vkhorikov www.enterprisecraftsmanship.com

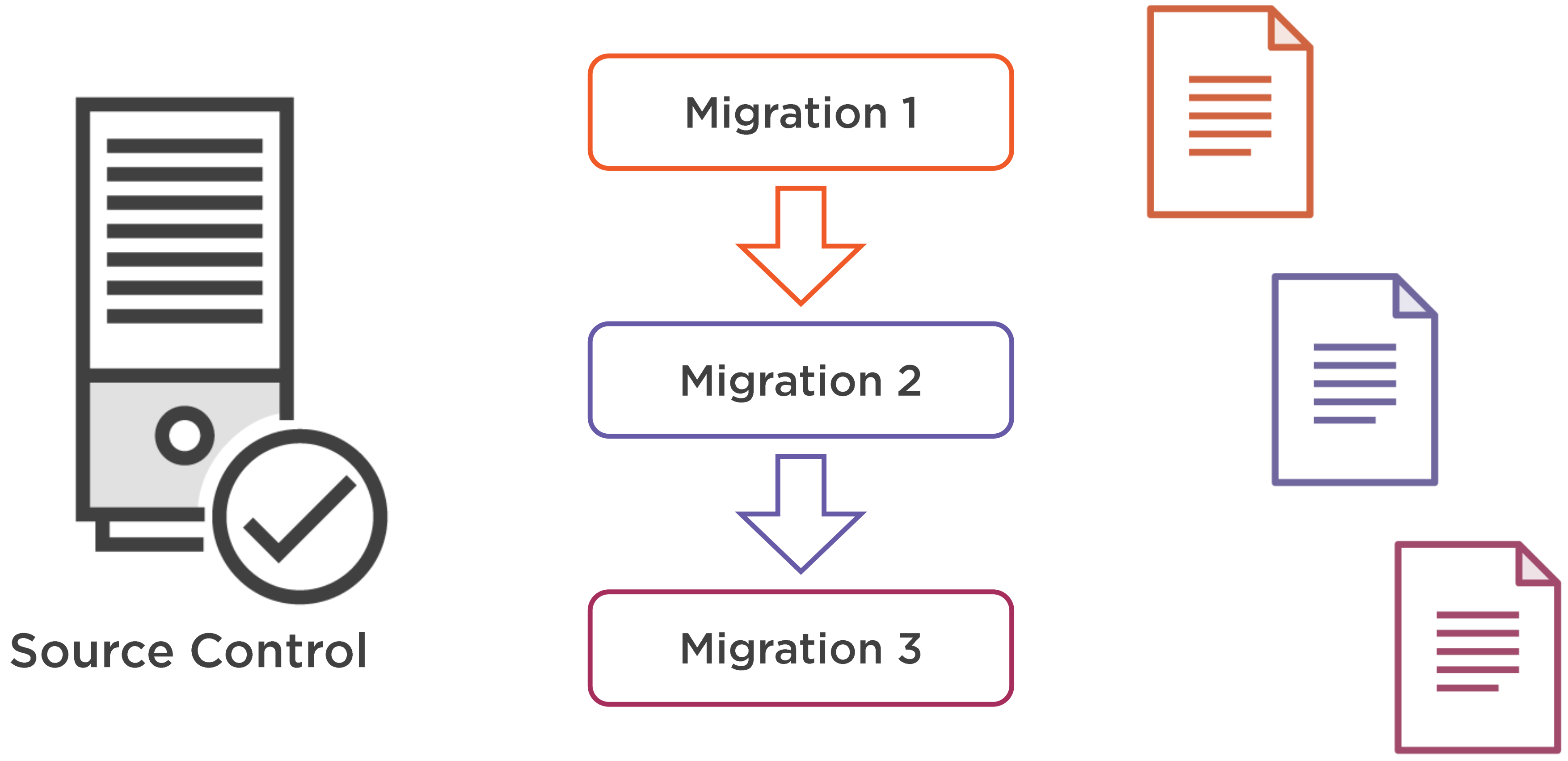
Outline

Fluent Migrator

Refactoring

Compare the two approaches

Introducing the Migration-based Approach



Introducing the Migration-based Approach

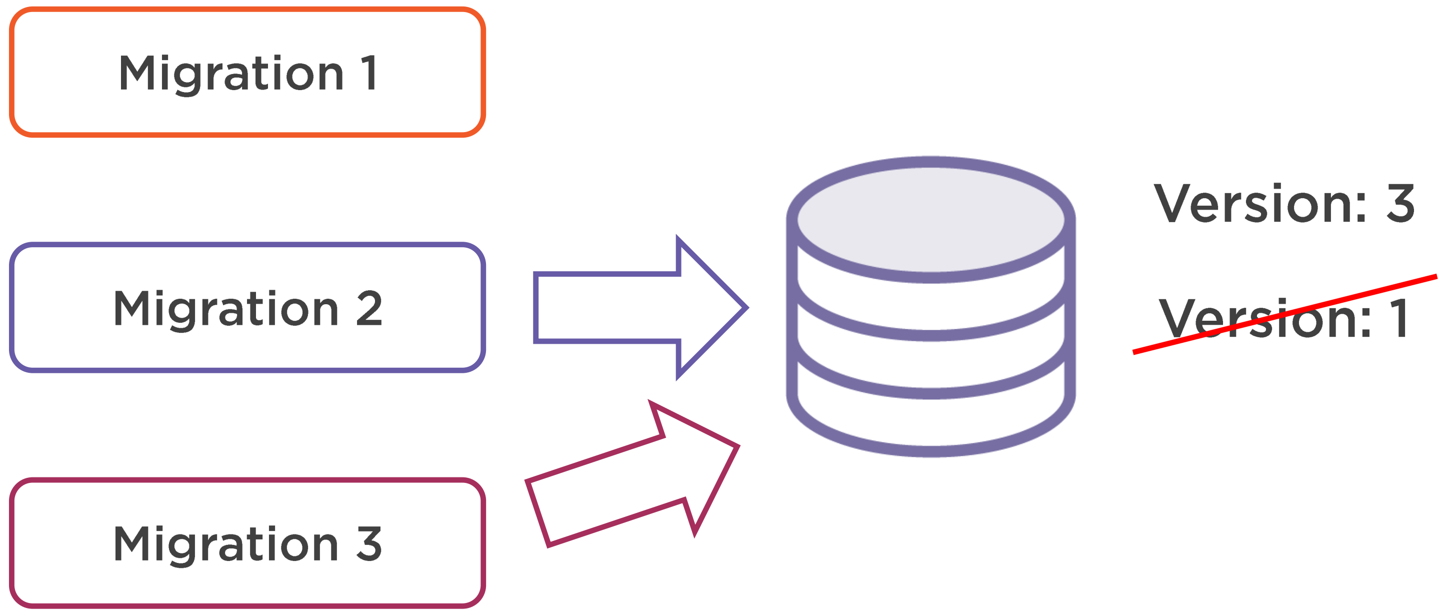
01_Initial.sql:

```sql
CREATE TABLE [dbo].[User] (
    [UserID] BIGINT NULL PRIMARY KEY
);
```

02_AddNameColumn.sql:

```sql
ALTER TABLE [User]
ADD Name nvarchar(200) NULL
```

Introducing the Migration-based Approach

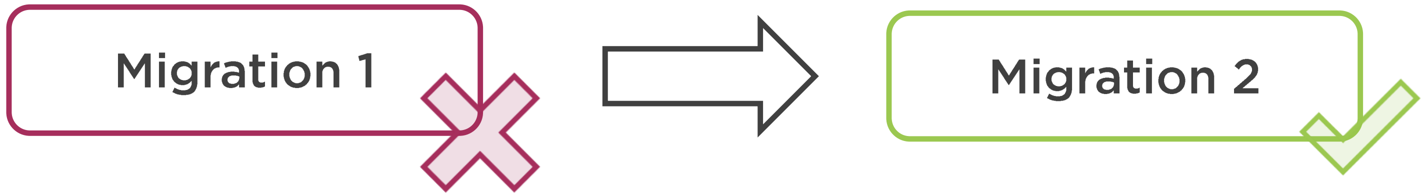


✓ **Great predictability**

Introducing the Migration-based Approach

Migrations must become immutable after deployment

Introducing the Migration-based Approach



✓ **Rolling forward**

✓ **Change only in case of a data loss**

Introducing Fluent Migrator

```
[Migration(2)]
public class Initial : Migration {
    public override void Up() {
        Create.Table("User")
            .WithColumn("UserID").AsInt64().NotNullable().PrimaryKey()
            .WithColumn("Name").AsString(200).NotNullable();
    }

    public override void Down() {
        Delete.Table("User");
    }
}
```

⬅ **Not for production**

✓ **Easily readable**

✓ **Strongly typed**

Recap: Creating an Initial Version

- Got started with the migration-based approach
- Database of version 1

- Persisted migrations
- Migrations are atomic

Entity Framework Code First Migrations

Recap: Splitting the Name Column

No pre- and post-deployment steps

All required steps are implemented in a single place

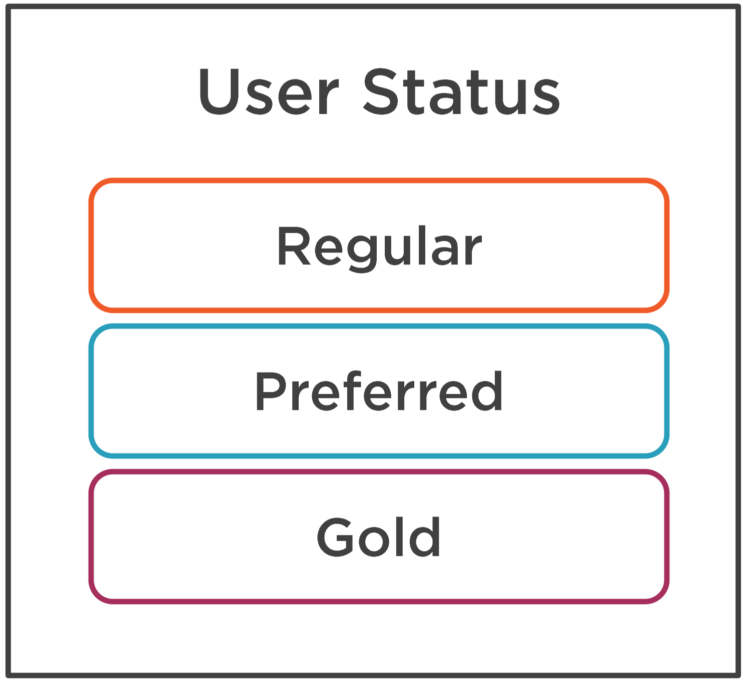
No need in the 2-step approach

Can define any transition regardless of its complexity

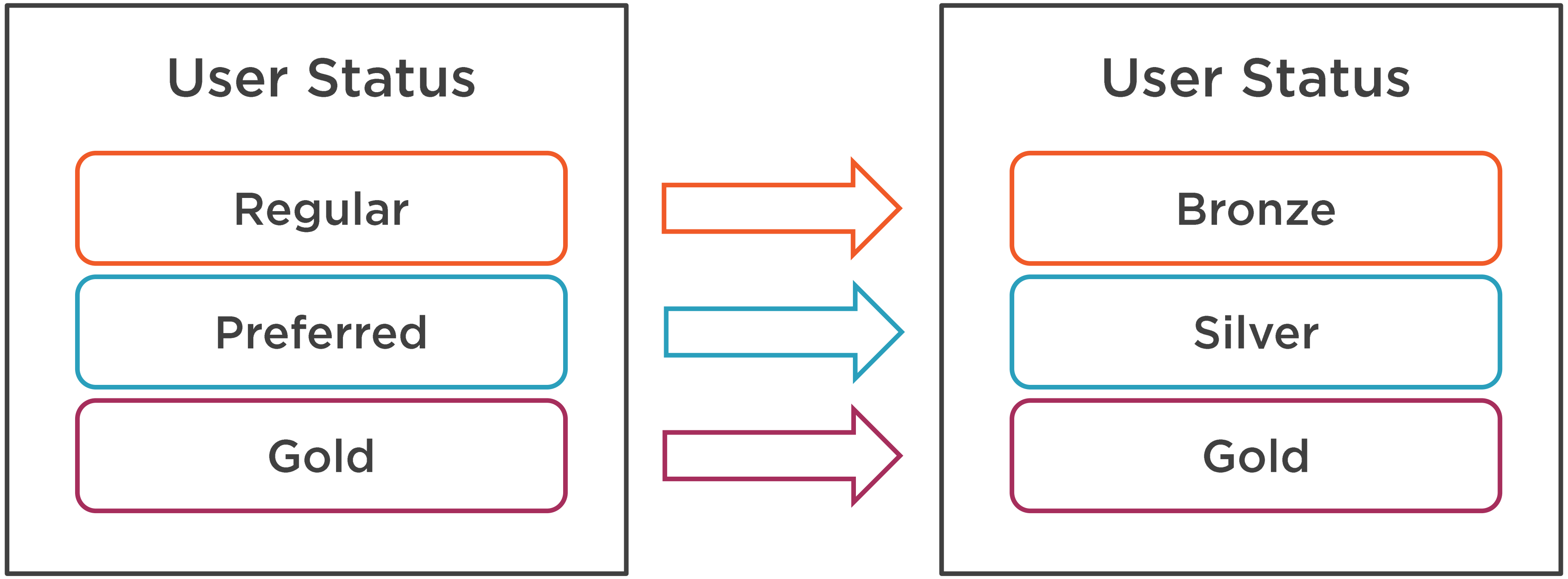
No need to track the production DB

Can build up several migrations one after another

Extracting the User Status Table



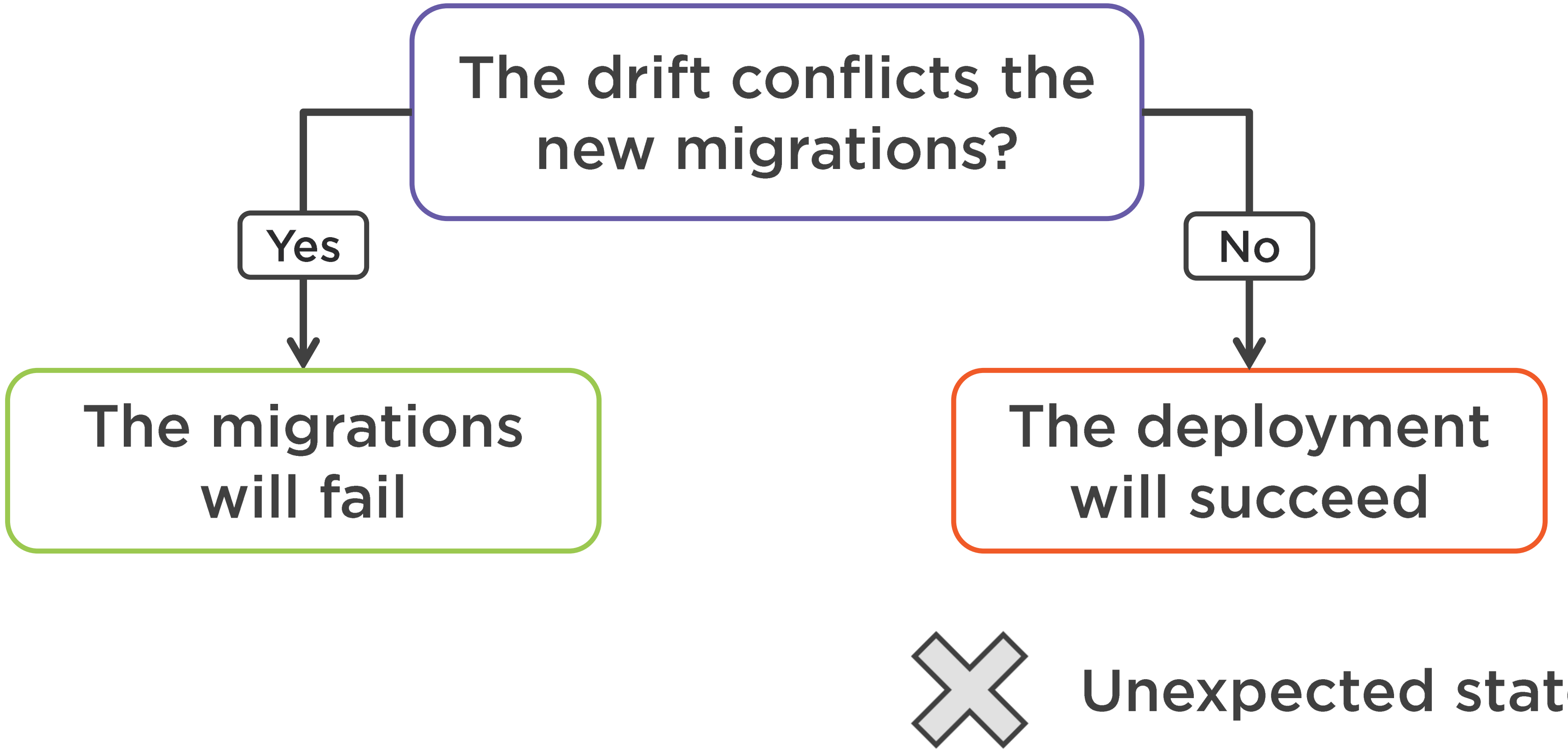
Changing Reference Data



Dealing with Database Drifts

Create a migration for any schema or reference data change in the database

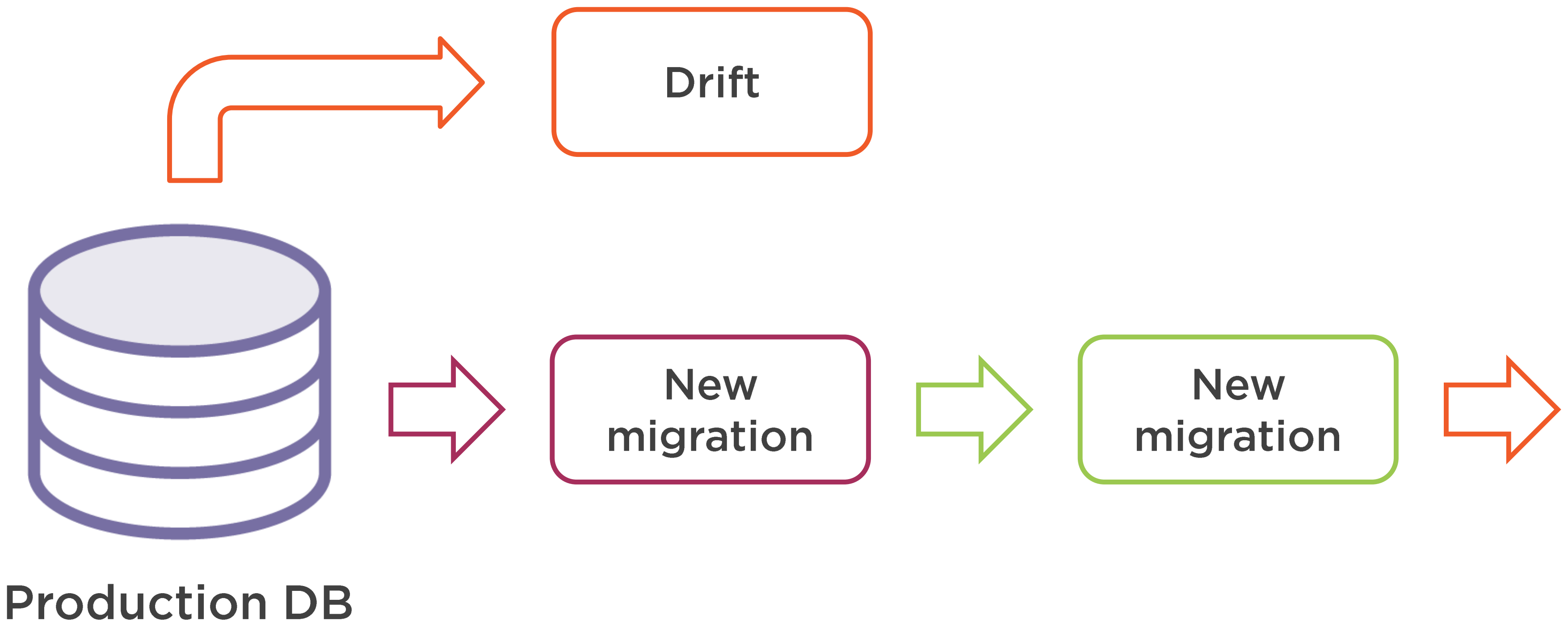
Dealing with Database Drifts


The drift conflicts the new migrations?
Yes
No
The migrations will fail
The deployment will succeed
Unexpected state

Dealing with Database Drifts

✓ **Always compare production and development databases**

Non-conflicting Drift

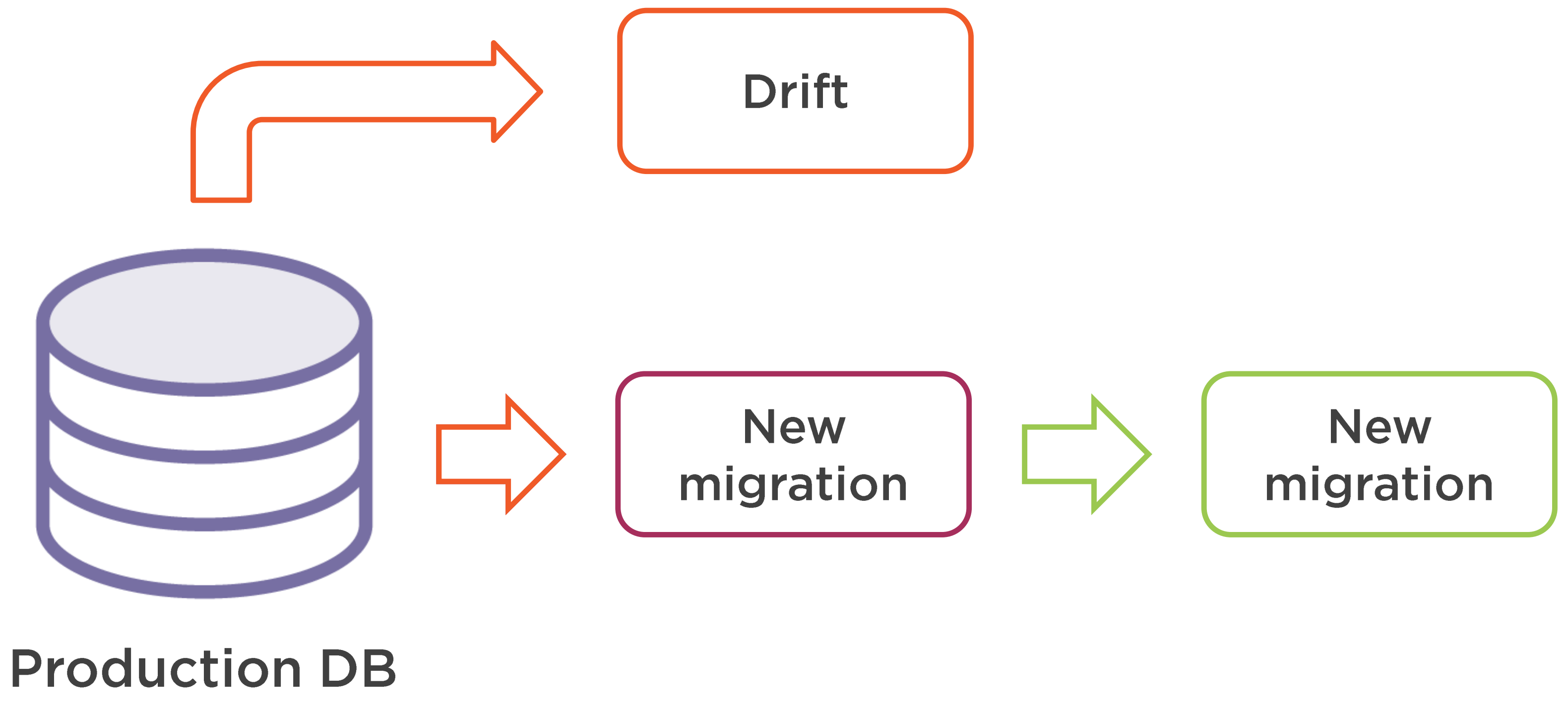


✓ **Make the drift-migration idempotent**

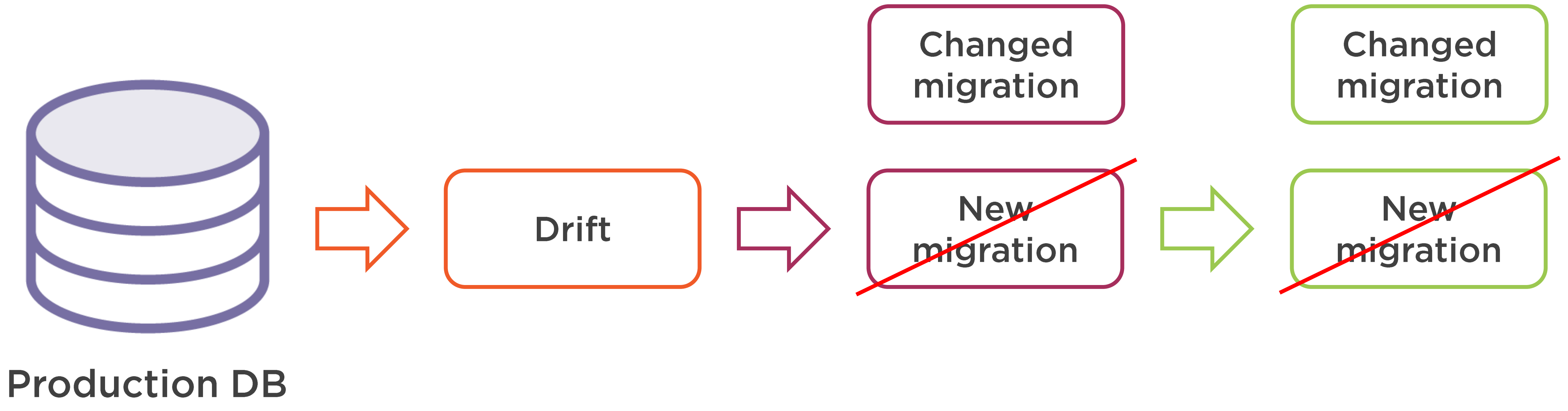
Non-conflicting Drift

```
if (!Schema.Table("User").Index("IX_User_Email").Exists())
{
    Create.Index("IX_User_Email")
        .OnTable("User")
        .InSchema("dbo")
        .OnColumn("Email")
        .Unique();
}
```

Conflicting Drift

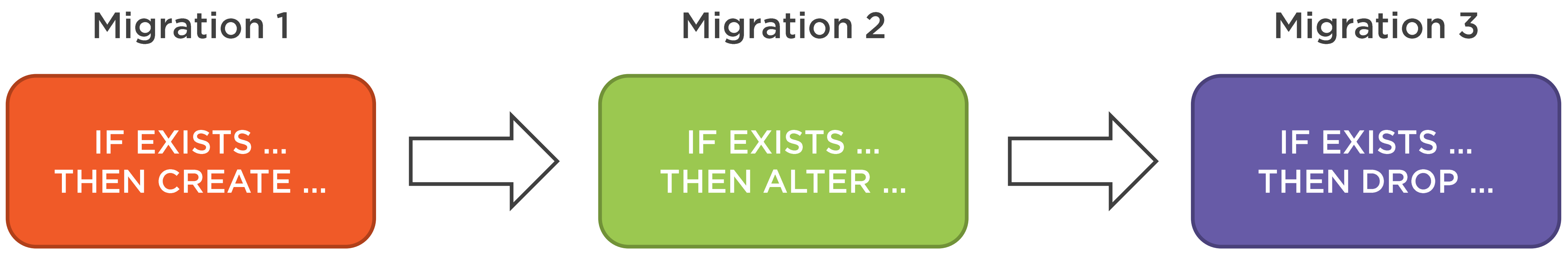


Conflicting Drift



✓ Make sure everyone gets the updated migrations

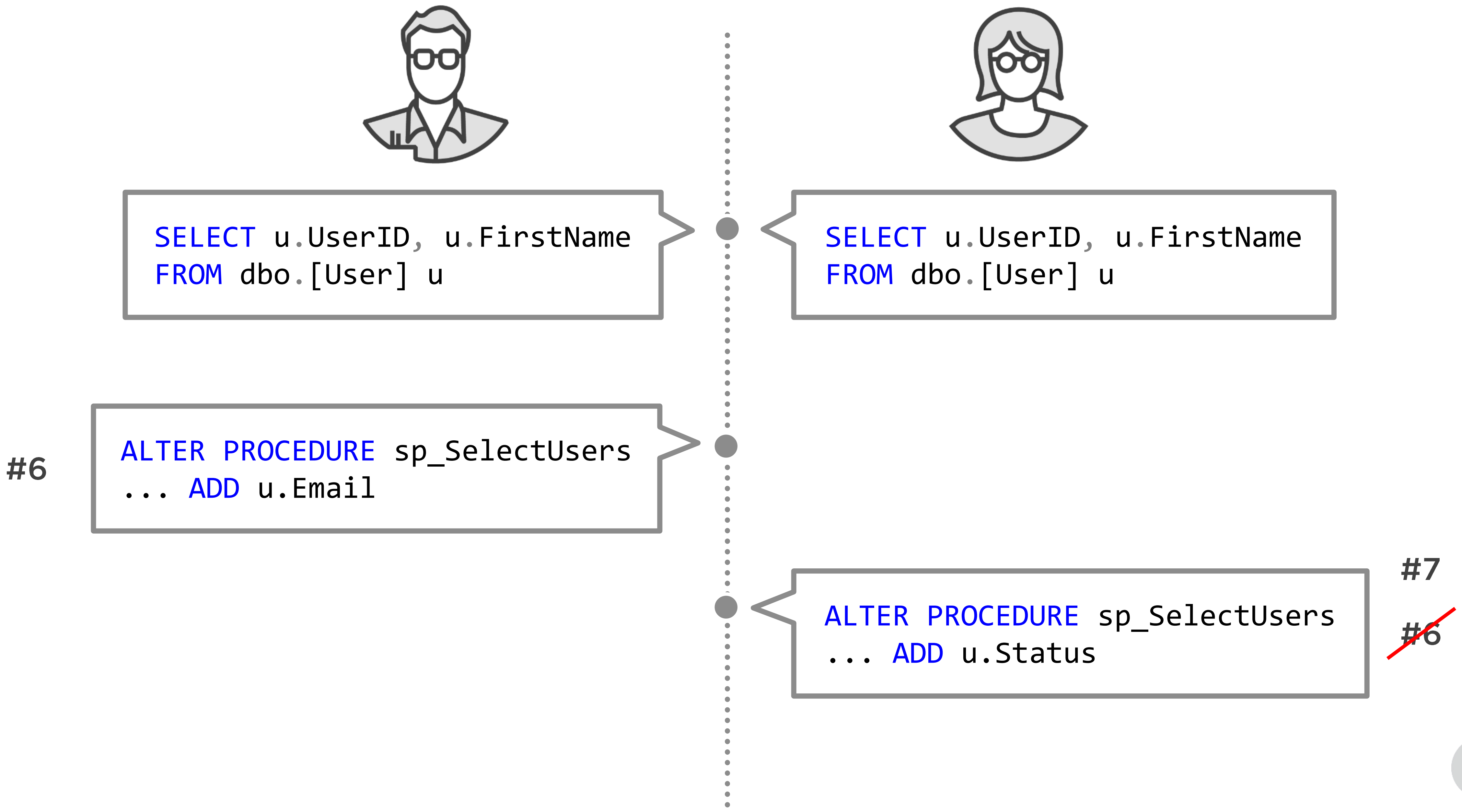
Dealing with Database Drifts



✓ **Make all migrations idempotent**

△ **Only if drifts are frequent**

Handling Merge Conflicts



```sql
SELECT u.UserID, u.FirstName
FROM dbo.[User] u
```

```sql
SELECT u.UserID, u.FirstName
FROM dbo.[User] u
```

```sql
ALTER PROCEDURE sp_SelectUsers
... ADD u.Email
```

```sql
ALTER PROCEDURE sp_SelectUsers
... ADD u.Status
```

Handling Merge Conflicts

Resolving conflicts is a tedious process

- ✓ Create as few branches as possible
- ✓ Conflicting tables are easier to spot

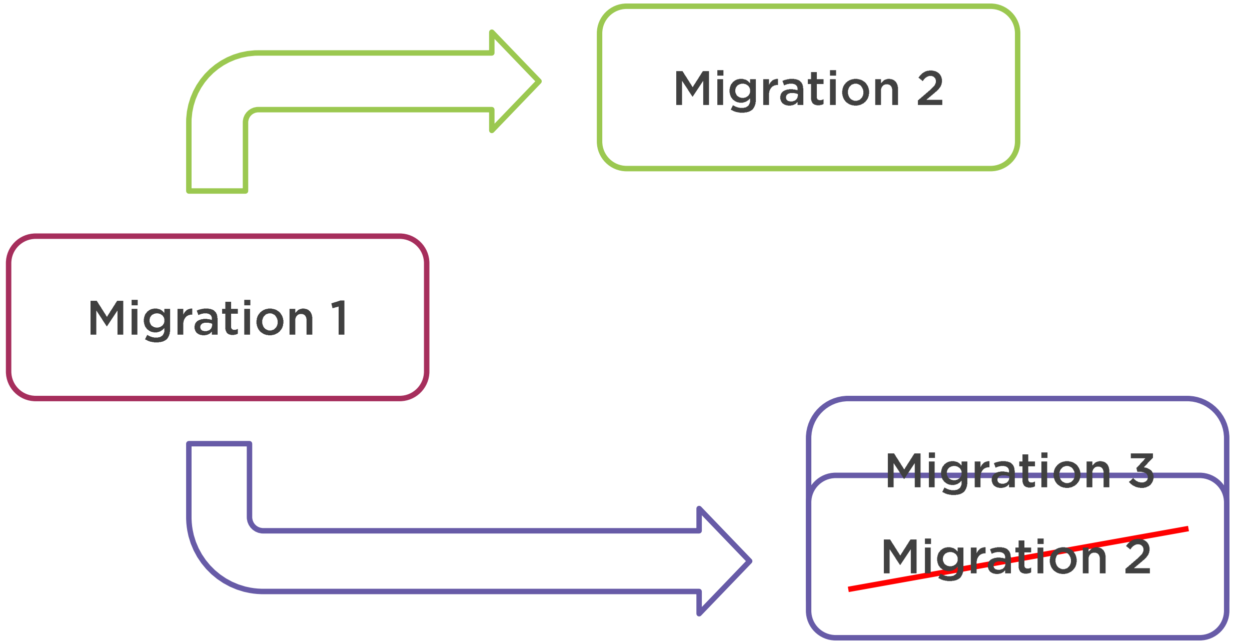
Enumerating Migrations: Numbers or Timestamps?

Numbers vs **Timestamps**

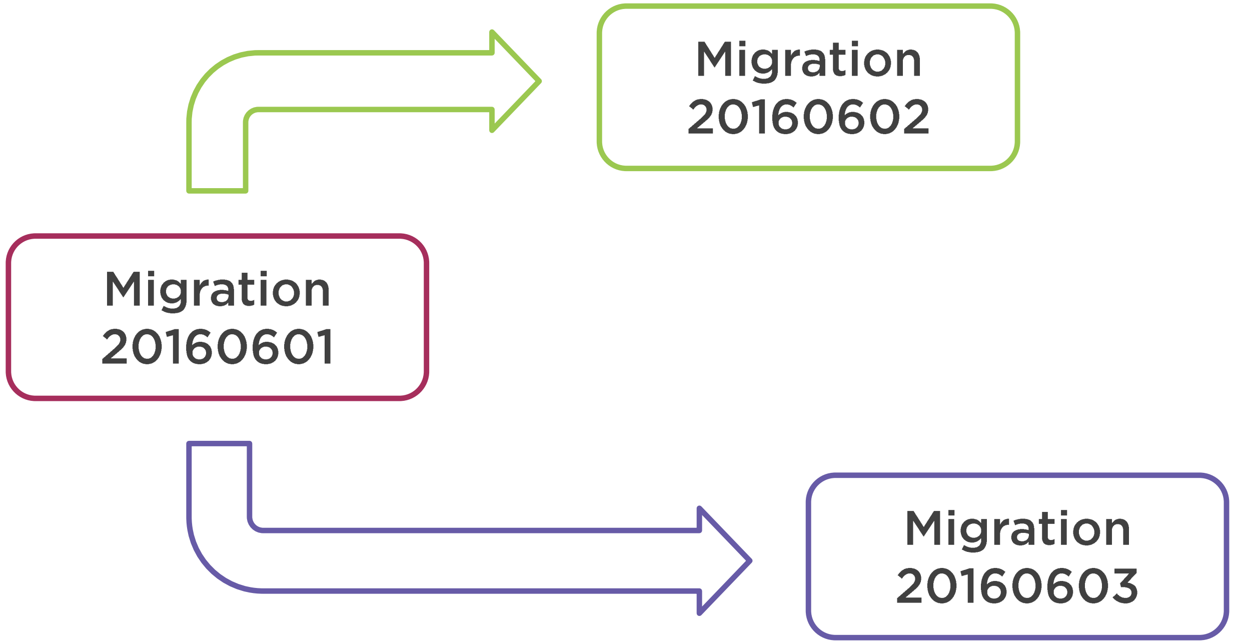
```
[Migration(1)]
public class MyMigration
{
}
```

```
[Migration(201606010420)]
// June, 1 2016 @ 4:20
public class MyMigration
{
}
```

Enumerating Migrations: Numbers or Timestamps?



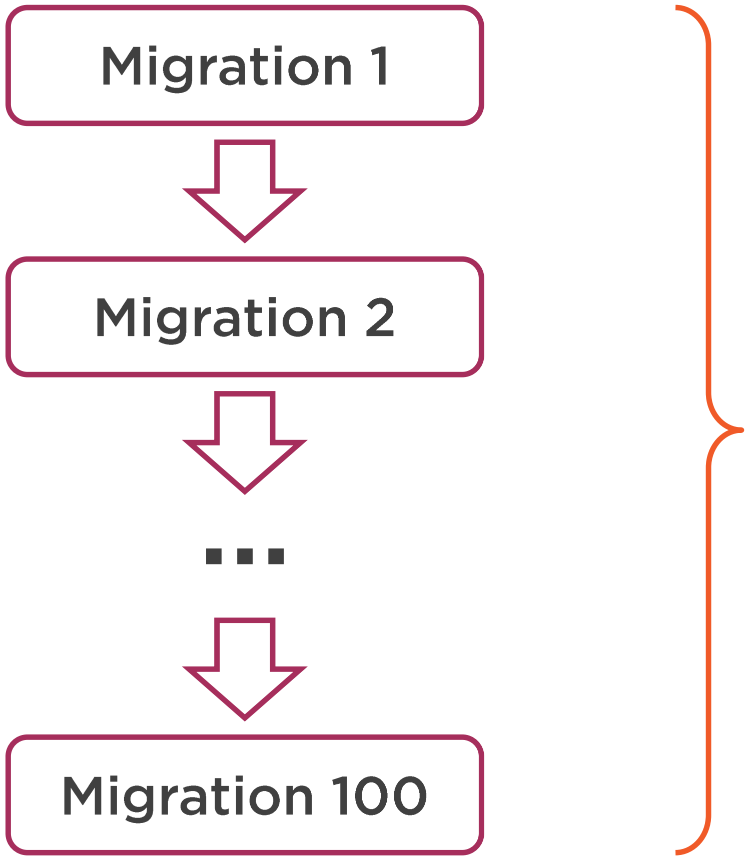
Enumerating Migrations: Numbers or Timestamps?



✓ Stick to numbers instead

✓ Numbers make a conflict explicit

What to Do with Ever Increasing Number of Migrations?



100_Base.sql

```
CREATE TABLE dbo.User
CREATE TABLE dbo.UserStatus
CREATE TABLE dbo.Company
CREATE STORED PROCEDURE ...
```

State-based vs. Migration-based Database Delivery

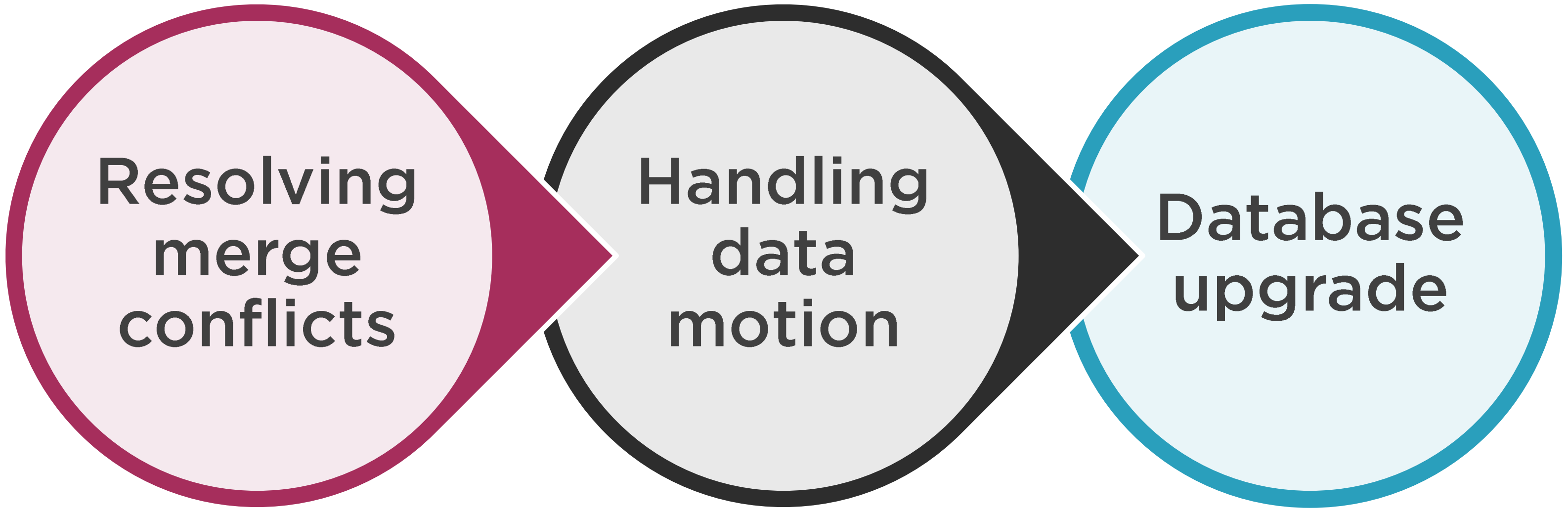
State-based approach

Migration-based approach

✓ Makes state explicit

✓ Makes transitions explicit

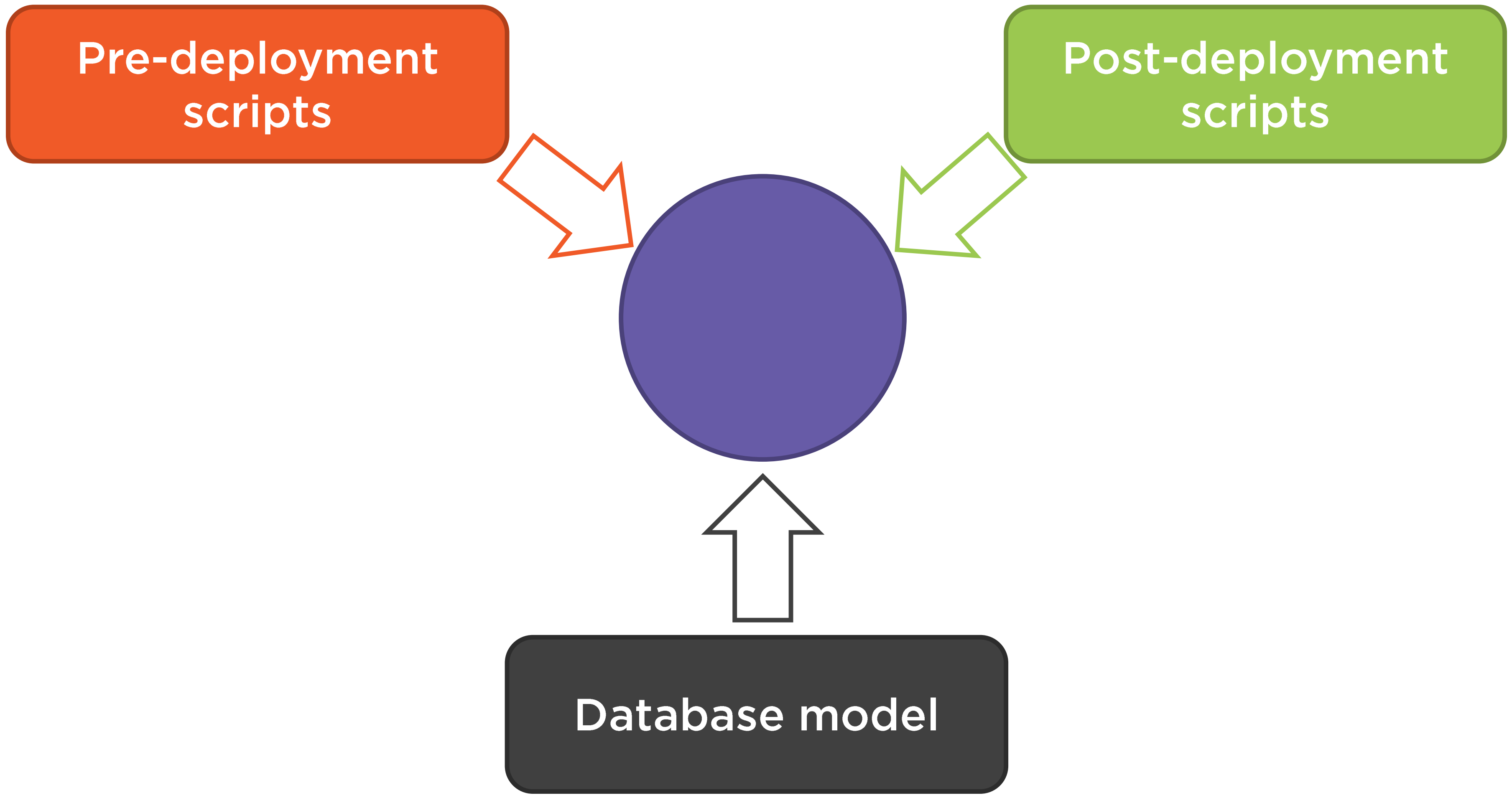
State-based vs. Migration-based Database Delivery



State-based vs. Migration-based Database Delivery

State-based	Migration-based
✓ Easy to handle merge conflicts	✗ Hard to handle merge conflicts
✗ No easy way to implement data motion	

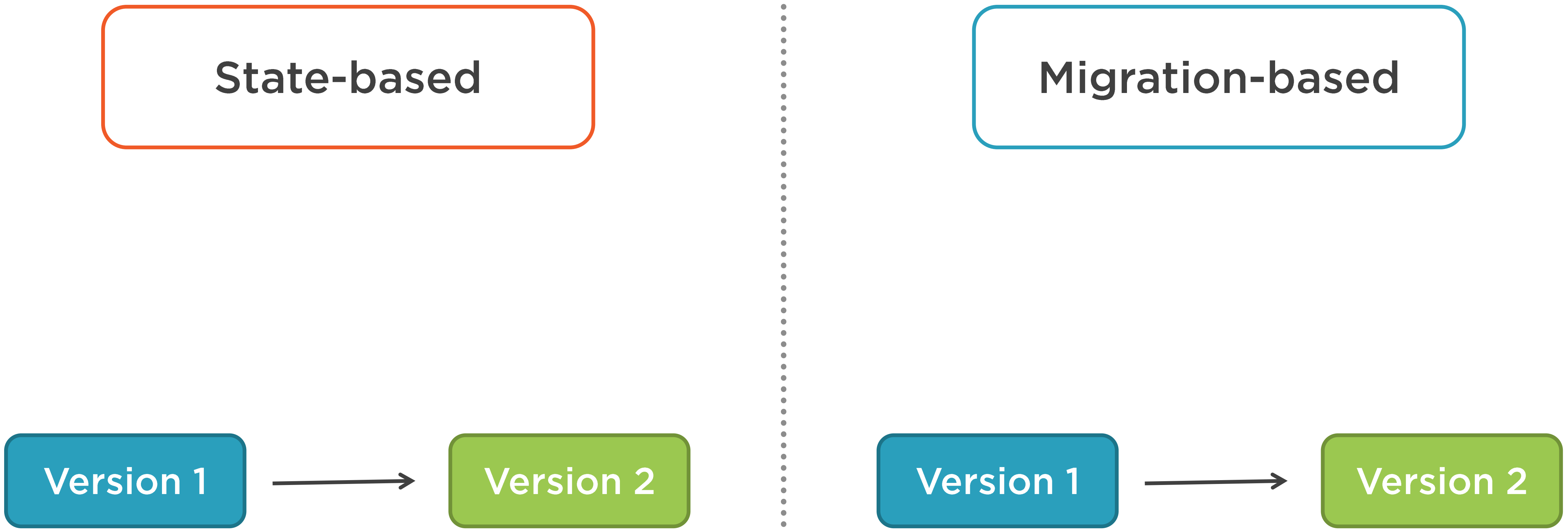
Data Motion with the State-based Approach



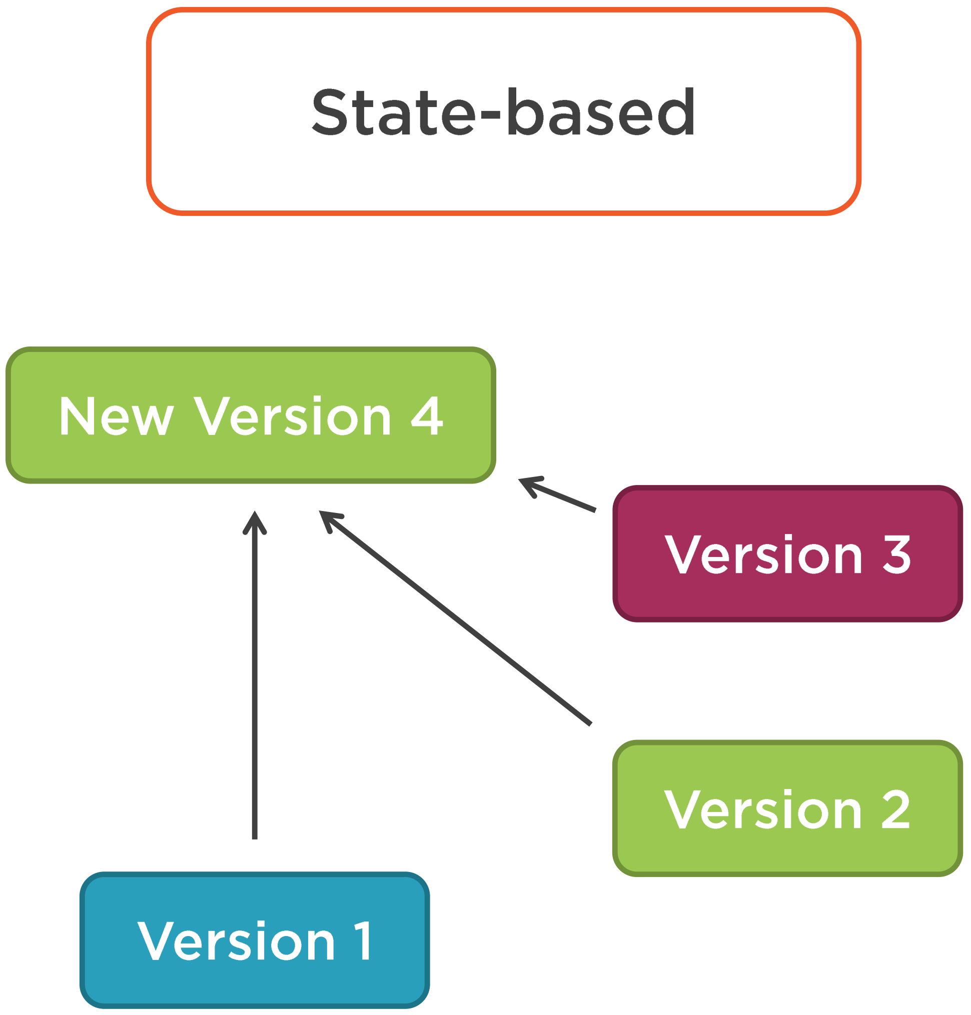
State-based vs. Migration-based Database Delivery

State-based	Migration-based
✓ Easy to handle merge conflicts	✗ Hard to handle merge conflicts
✗ No easy way to implement data motion	✓ Data motion is coherent with schema changes

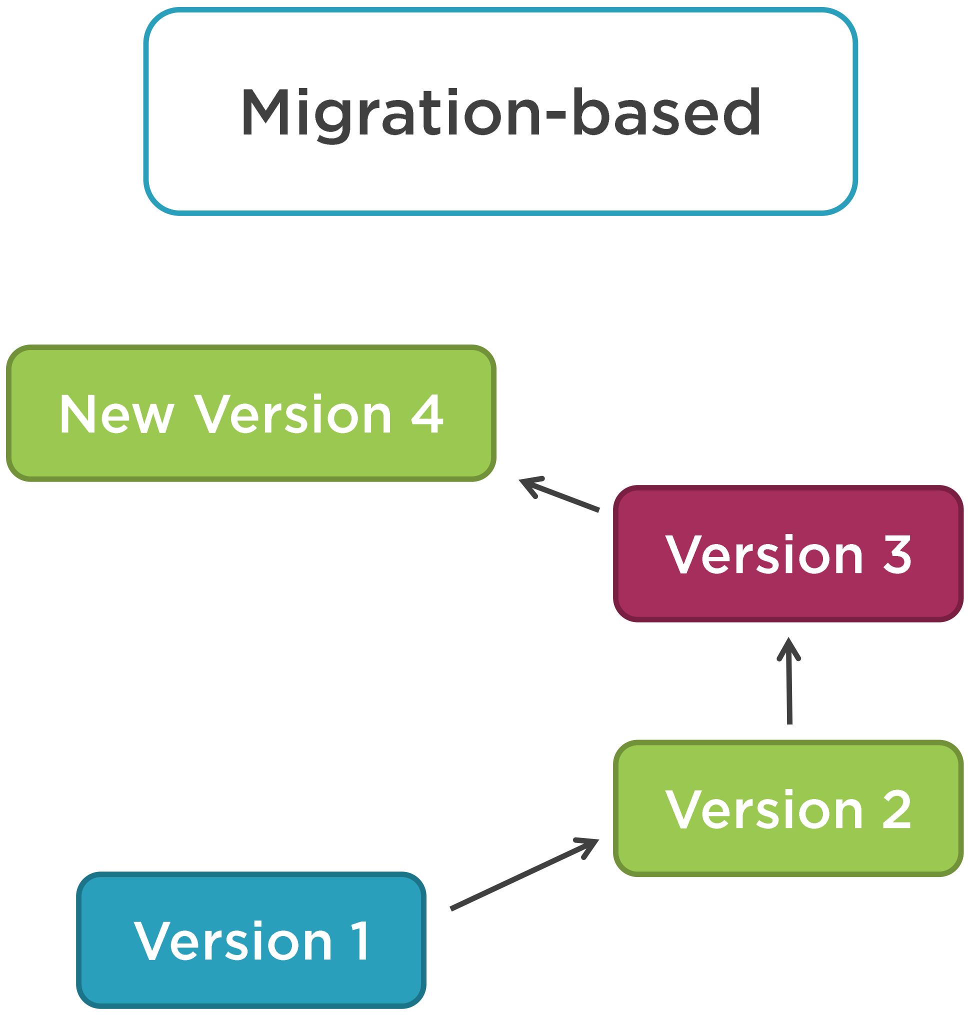
Upgrading the Target Database



Upgrading the Target Database



Number of transitions to test: 3



Number of transitions to test: 1

Upgrading the Target Database

State-based	Migration-based
✗ 2-step approach for complex refactorings	✓ No restrictions on complexity of refactorings

State-based vs. Migration-based Database Delivery

State-based	Migration-based
✓ Easy to handle merge conflicts	✗ Hard to handle merge conflicts
✗ No easy way to implement data motion	✓ Data motion is coherent with schema changes
✗ Hard to deal with multiple production databases	✓ No additional effort with multiple production DBs
✓ Easier to work with multiple branches	✓ Easier to bring a database to some particular state

State-based vs. Migration-based Database Delivery

The elephant in the room: Continuous Delivery for Databases

https://vimeo.com/131637362

What Approach to Choose?

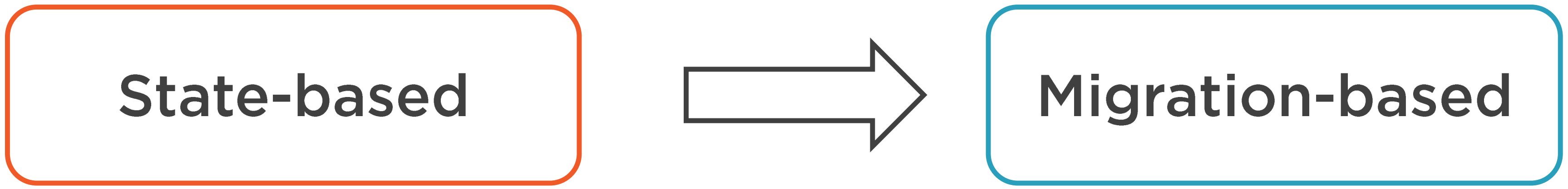
State-based if:

- A lot of logic in the database
- Large distributed team

Migration-based if:

- Not much logic in the database
- Multiple production databases
- Small local team

Transitioning from One Approach to the Other



✓ Create a base line script

Transitioning from One Approach to the Other

✓ **State-based approach on early stages**

No need to create migrations if no data is in production yet

✓ **Migration-based approach when in production**

Allows for handling data motion and refactorings

Combining the Two Approaches

Is it possible to combine them

Combining the Two Approaches

Conflicts in tables

✓ Spotted when re-creating the DB

Conflicts in stored procedures

✓ Spotted by the source control

Moving SQL Code to the Application Layer

Forgo using stored procedures

✓ No logic in the database

Moving SQL Code to the Application Layer

```
public class UserRepository
{
    public IReadOnlyList<UserDto> GetAll()
    {
        string text = @"
            SELECT u.UserID, u.FirstName, u.LastName, u.Email, s.Name [Status]
            FROM dbo.[User] u
            INNER JOIN dbo.UserStatus s ON u.StatusID = s.UserStatusID";

        /* Transform the results of the query to DTOs */
    }
}
```

❌ **Not applicable to database functions**

Summary

The migration-based approach to database delivery

- Makes transitions explicit
- Try to keep migrations immutable

Refactoring using Fluent Migrator

Dealing with database drifts

- Always compare the production DB with the development DB
- Make migrations idempotent

Prefer numbers over timestamps for versions

When too many migrations, rebase them

Summary

Pros and cons of the state-based and migration-based approaches

- State is better for dealing with merge conflicts
- Migrations are better for data motion and multiple production databases

Choose the state-based approach if:

- You have a large distributed team
- The database contains a lot of logic

Choose the migration-based approach if:

- The database structure changes often
- You have a small local team
- Have multiple production databases

How to combine them together

In the Next Module

Building your own database versioning tool