Patterns for Monolith to Microservice Migration

Which pattern to choose when?

Saurabh Dashora

Moving from a monolithic architecture—where all functionality is bundled together—into a microservices architecture isn't a flip of a switch. It's a complex journey that, if done recklessly, can cause more harm than good.

That's why smart teams follow specific transition patterns. These patterns help reduce risk, maintain stability, and allow the system to evolve gradually rather than completely rebuild.

Here are 4 such patterns that can help you:

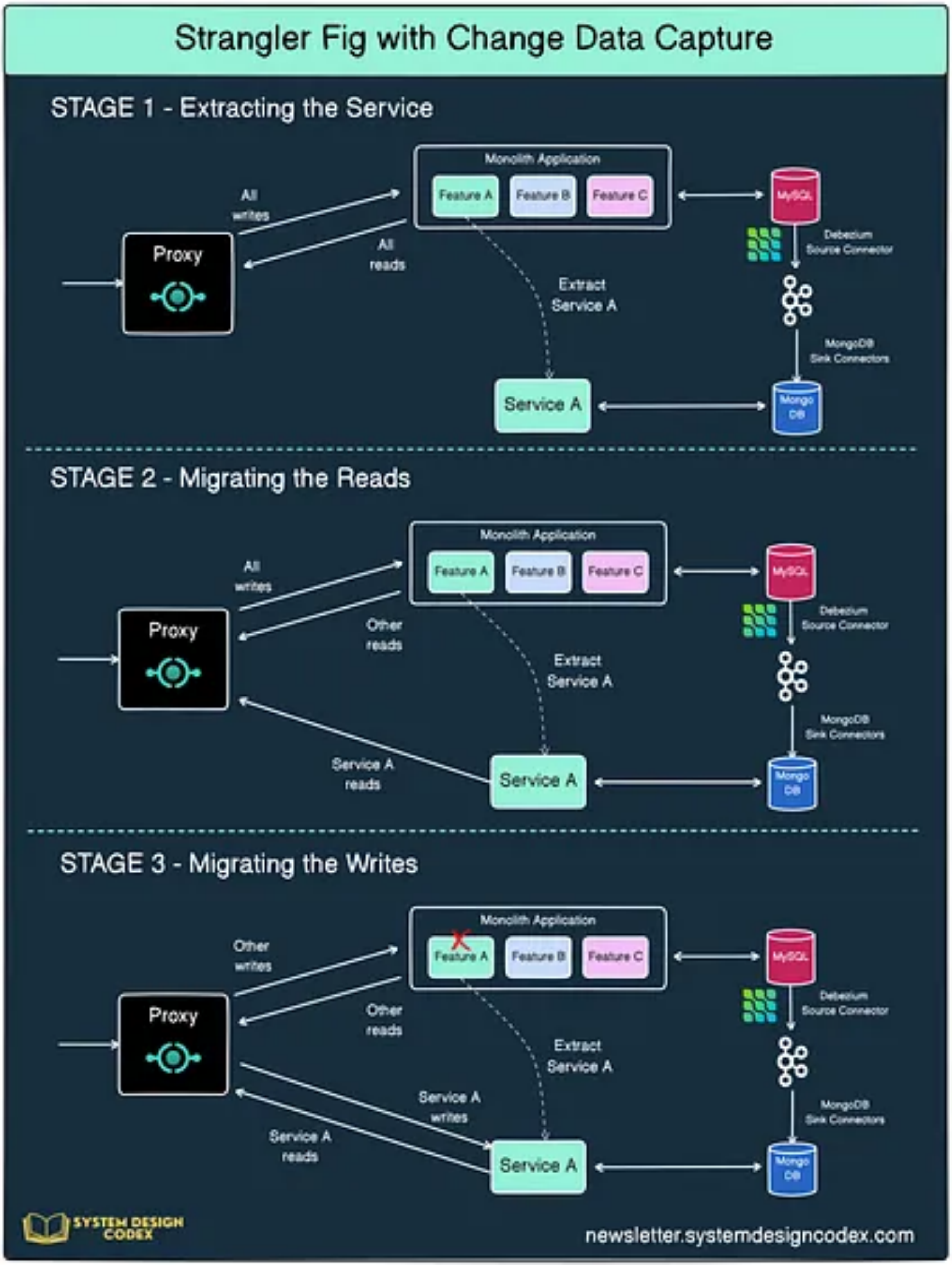
1 - Strangler Fig Pattern

The Strangler Fig Pattern is perhaps the most famous strategy for modernizing monolithic systems. It's inspired by how a strangler fig tree grows around an existing tree, eventually replacing it.

How It Works:

- An API gateway or proxy is placed in front of the monolith.
- New features or refactored versions of existing features are developed as independent microservices.
- The gateway routes requests:
    - To the monolith for old functionality.
    - To the new microservices for updated functionality.
- Over time, as more functionality is moved out, the monolith shrinks

until it can be retired.



You can play around with the image on Eraser.io

Why It’s Powerful:

- Incremental migration with minimal disruption.
- Reduces the big-bang rewrite risk.
- You can prioritize extracting high-value or high-risk components first.

> Real-world use: Many enterprises use this approach when dealing with legacy systems where complete rewrites would be risky and expensive.

2 - Parallel Run Pattern

The Parallel Run Pattern involves running the old monolith and new microservices together for a period of time. Instead of replacing immediately, you redirect a portion of traffic to microservices to validate their behavior.

How It Works:

- Both the monolith and the new microservices are deployed simultaneously.
- Certain user requests or workflows are mirrored or split-routed:
  - Some go to the monolith.
  - Some go to the microservices.
- The output or behavior is compared to ensure that the new system is functionally equivalent.

You can play around with the diagram on Eraser.io

Why It’s Useful:

- Catch errors early before full migration.
- Allows thorough real-world testing.
- You can gradually increase traffic to the microservices as confidence builds.

> Pro Tip: Use feature flags and automated comparison testing to make this process smoother.

3 - The Collaborator Pattern

The Collaborator Pattern is a clever way to augment the monolith without touching its core logic immediately. It allows you to extend the system by attaching microservices around it.

How It Works:

- Instead of replacing a functionality inside the monolith, you decorate or wrap existing modules with microservices that add new features.
- The monolith still handles the original core behavior.
- Microservices add enhanced behavior, enrich responses, or intercept and modify calls.

Why It's Valuable:

- Minimal disruption to the monolith.
- Let's you introduce new functionality faster without waiting for core refactoring.
- Helps test microservice integration with live traffic early.

> **Example:** Suppose you have a monolith that processes orders, but you want to add a new discount service. Instead of modifying the order logic, you add a microservice that "decorates" order responses with discount information.

4 - Change Data Capture (CDC)

One of the biggest challenges during a transition is data synchronization. Change Data Capture (CDC) provides a solution by streaming real-time database changes from the monolith to the new microservices.

How It Works:

- Tools like Debezium, Kafka Connect, or AWS DMS monitor the monolith's database transaction logs.

- Whenever a change happens (insert, update, delete), the change is captured and streamed to interested microservices.
- This allows microservices to react in near real-time without needing tight coupling to the monolith's database.

You can play around with the diagram on Eraser.io

Why It's Critical:

- Enables event-driven architectures during the transition.
- Allows microservices to maintain their own data stores based on monolith changes.
- Reduces reliance on synchronous API calls to the monolith.

> Important Tip: Be cautious about schema evolution; changes to the monolith database schema must be carefully coordinated with CDC consumers.

How These Patterns Work Together

Transitioning from a monolith isn't about choosing just one pattern—it's about orchestrating them together:

- Start with the Strangler Fig to route traffic intelligently.
- Run critical paths in parallel to build trust in the new system.
- Decorate existing functionalities to add features without destabilizing the core.
- Stream changes using CDC to gradually separate data ownership.

So - have you used any of these patterns?

Leave a comment

Shoutout

Here are some interesting articles I've read recently:

- What if this happens twice by
  Raul Junco
- From Zero to Software Engineer: 100+ Resources I Wish I Had at 18 by
  Fran Soto
- False promises fatigue by
  Riccardo Causo
- How to Better Organize Your React Component Files by
  Petar Ivanov

That's it for today! ☀️

Enjoyed this issue of the newsletter?

Share with your friends and colleagues.

Share