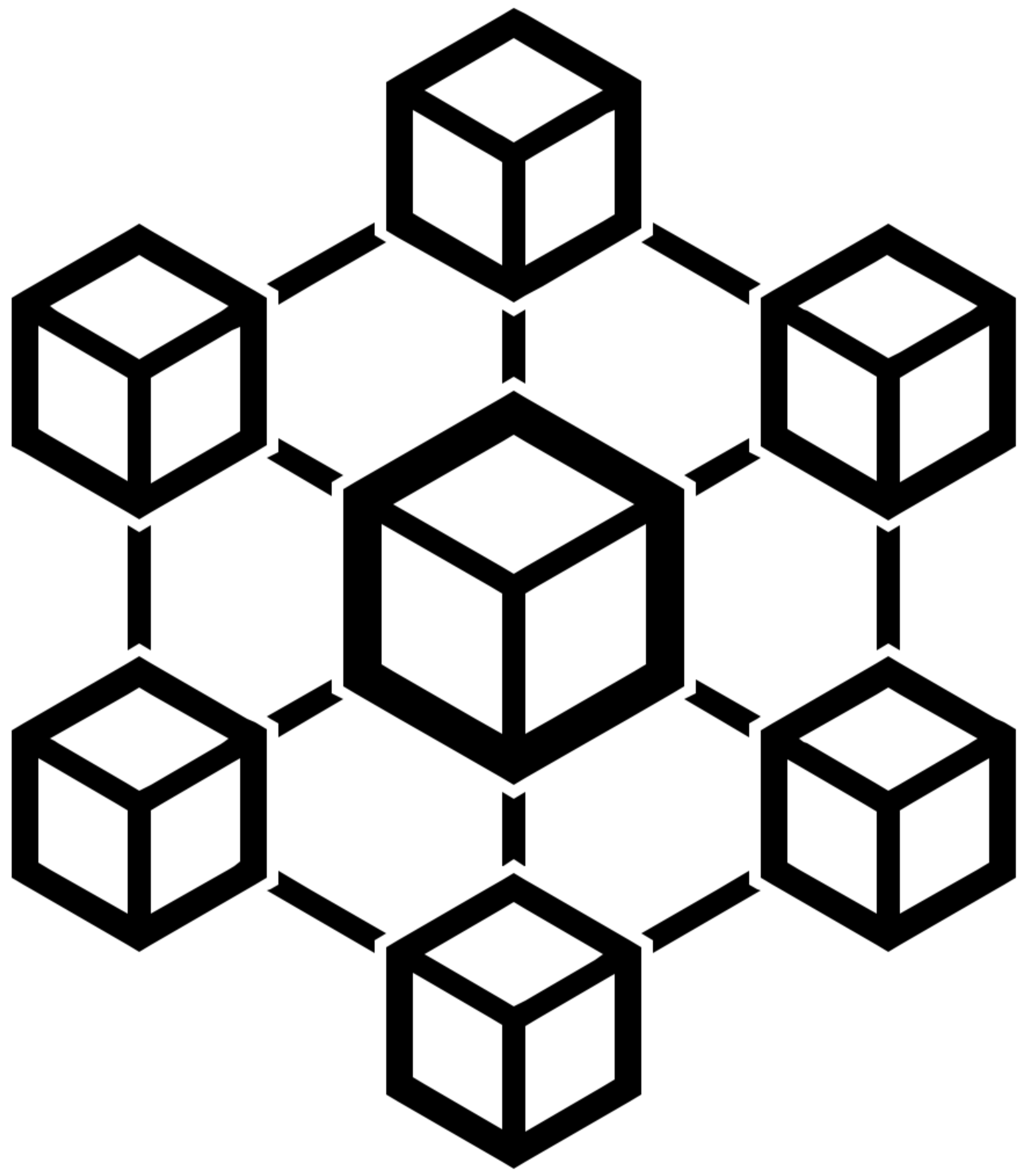
Microservices - Managing the chaos

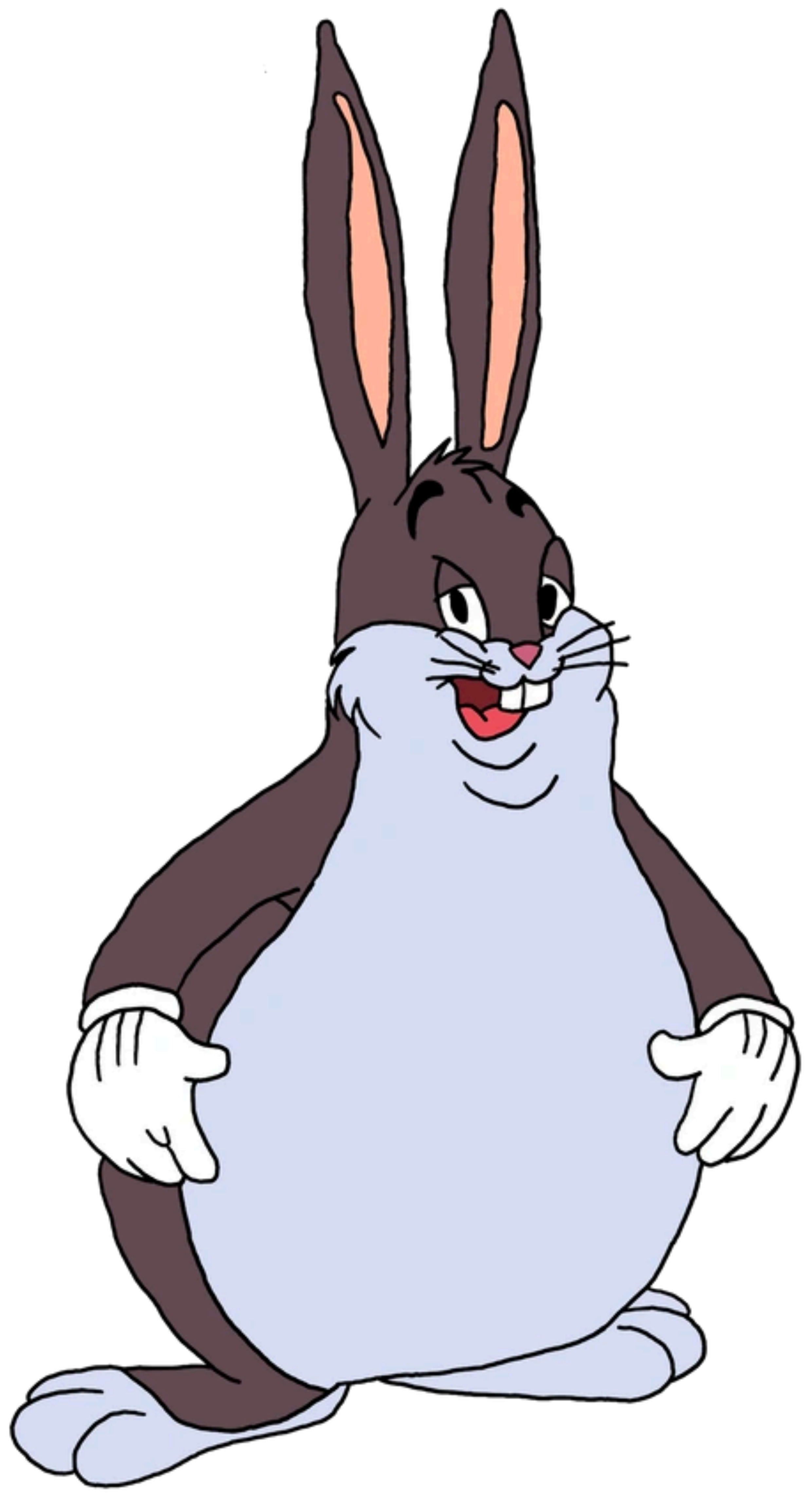
Eamon Scullion



Lets solve a real world problem!

Business Requirements

- The Business™ requires you to build an application for consuming a lot of carrots
- The UX team have put together some wireframes that look kind of familiar..



There's a few different ways that we can approach this..

Let’s apply a principle - Conway’s Law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure

Melvin E. Conway

What does Conway’s law mean for us?

- Teams who can communicate easily (small and collocated) tend to create monoliths

Traditional architecture: Monolith

- Teams who communicate through intermediaries (managers, architects etc.) tend to work on separate modules

Traditional architecture: “Modularith”

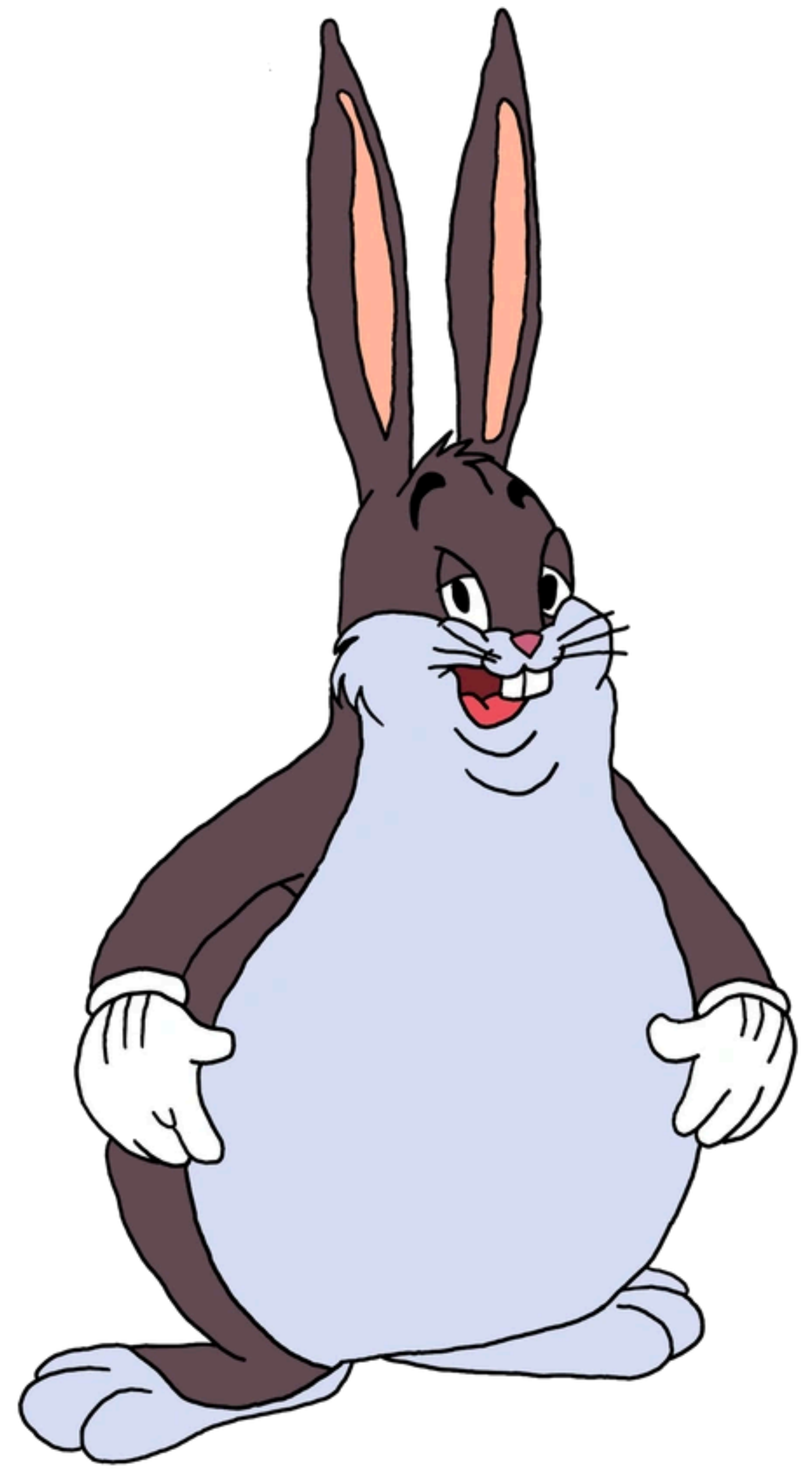
- Teams who communicate through hierarchies tend to work on complex structures. For example one core team, frontend teams, backend teams etc.

Traditional architecture: Microservices

Monoliths

How could we approach this with Monoliths?

- They are usually associated with legacy applications, but they aren’t inherently bad.
- Well suited for small teams who can “own” the whole application.
- Straight forward infrastructure to get started, allowing developers to deliver quickly (only one deployment artefact to worry about).
- Bottlenecks can be identified and separated as needed into separate modules, libraries, micro-services, serverless functions etc.



Problems with monoliths

- Separation and establishing boundaries may be a challenge if there is a lot of cross pollination (all code dependencies on all other code)
- Difficult to scale, inefficient as you need multiple instances of the same app rather than the specific bottlenecks
- As time goes on, monoliths grow and become harder and harder to maintain

Modulariths

How could we approach this with Modulariths?

- Modulariths is an approach to managing monoliths by having clearly defined and separated modules.
- Techniques like Domain-Driven Design are usually applied to establish boundaries.
- Allows easy separation into libraries, services etc in future. Modules can be deployed as separate services if required to scale independently
- Less complexity than a distributed system

Microservices

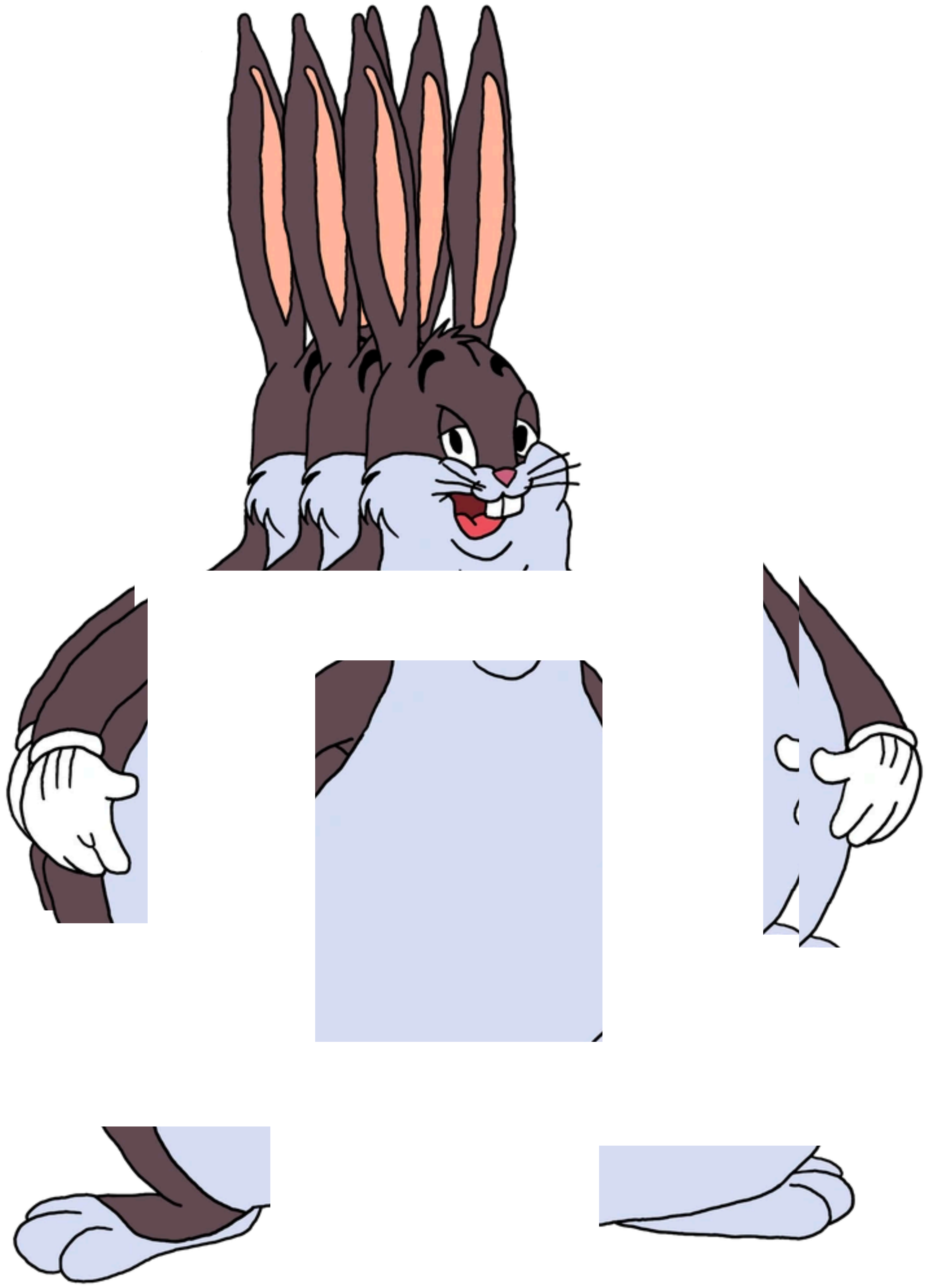
So.. what is a Microservices Architecture?

An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often a HTTP resource API.

Martin Fowler

Some of the benefits:

- The independent deployments enable independent development of the particular services.
- Communication is achieved through network calls (like HTTP), enabling polyglot programming - each service can use the best tool for the job.
- Flexibility for scaling individual services depending on load.
- Starting with microservices first is the easiest way to establish boundaries.



The biggest issue in changing a monolith into micro services lies in changing the communication pattern

Martin Fowler

Some of the problems

- While they allow you to deploy services independently, in reality, deployment and versioning is complex when you have a large amount of services.
- Large initial infrastructure investment required.
- Primarily communicates using network which is prone to latency issues.
- More integrations and failure points that you wouldn’t get in monoliths (e.g downtime in other services, unreliable networks etc).

How can we mitigate some of these problems?

1. Enable smoother communication using Contract testing

What is Contract testing?

- **Contract testing** is a technique for testing an integration point by checking each application in isolation to ensure the messages it sends or receives conform to a shared understanding that is documented in a “contract".
- For applications that communicate via HTTP, these "messages" would be the HTTP request and response, and for an application that used queues, this would be the message that goes on the queue.

Contract Testing

- Microservices require a different testing strategy to be effective.
- Microservices are meant to be able to be deployed individually.
- For developers to move at a rapid pace, they need assurance that they won’t break existing integrations.
- Contract tests assert that inter-application messages conform to a shared understanding that is documented in a contract.
- Without contract testing, the only way to ensure that applications will work correctly together is by using expensive and brittle integration tests.

What problem does this solve?

Look familiar?

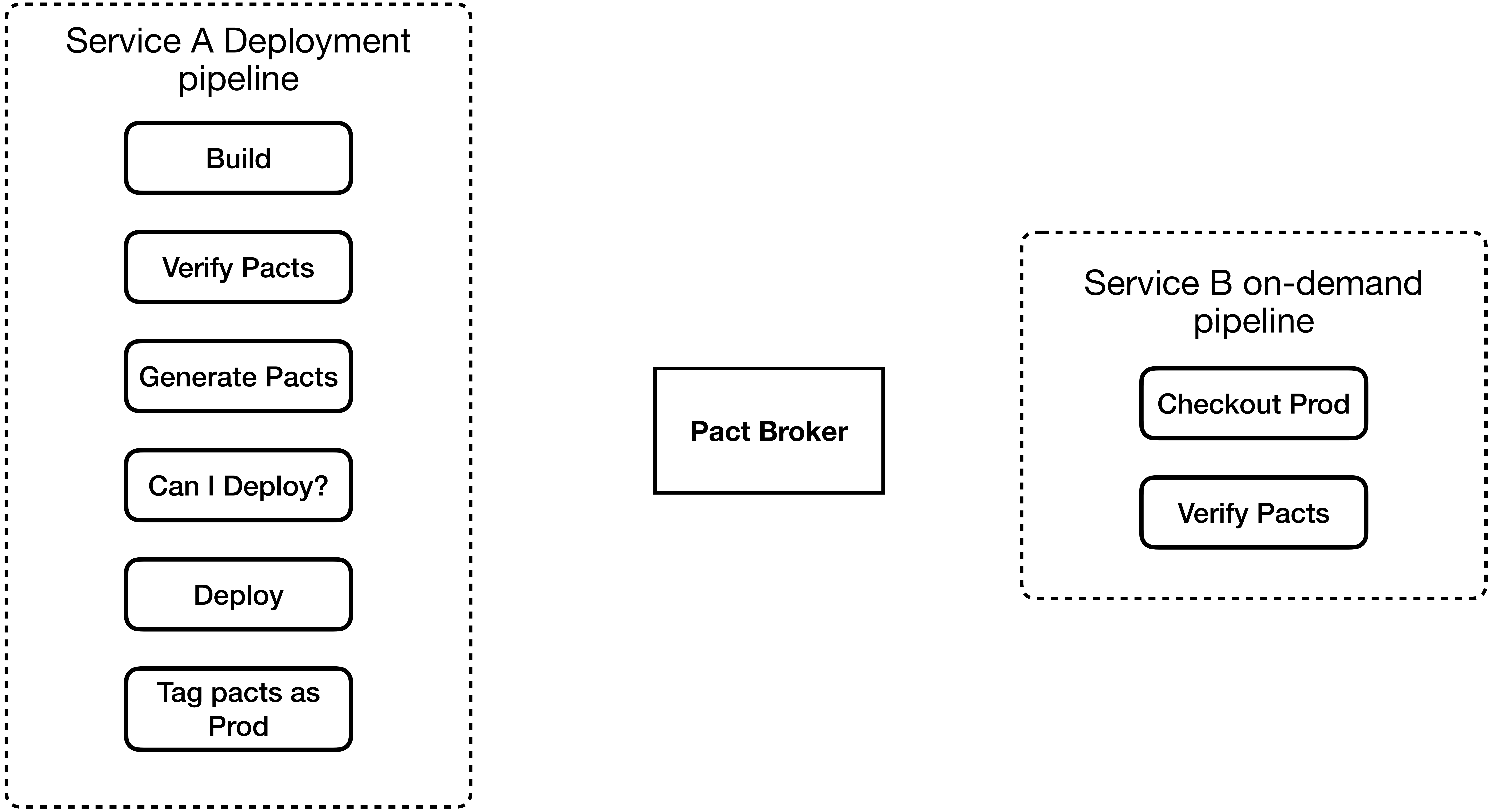
FRONTEND DEVELOPER
YOU CAN'T JUST CHANGE ENDPOINTS WHENEVER YOU WANT, IT KEEPS BREAKING OUR FRONTEND

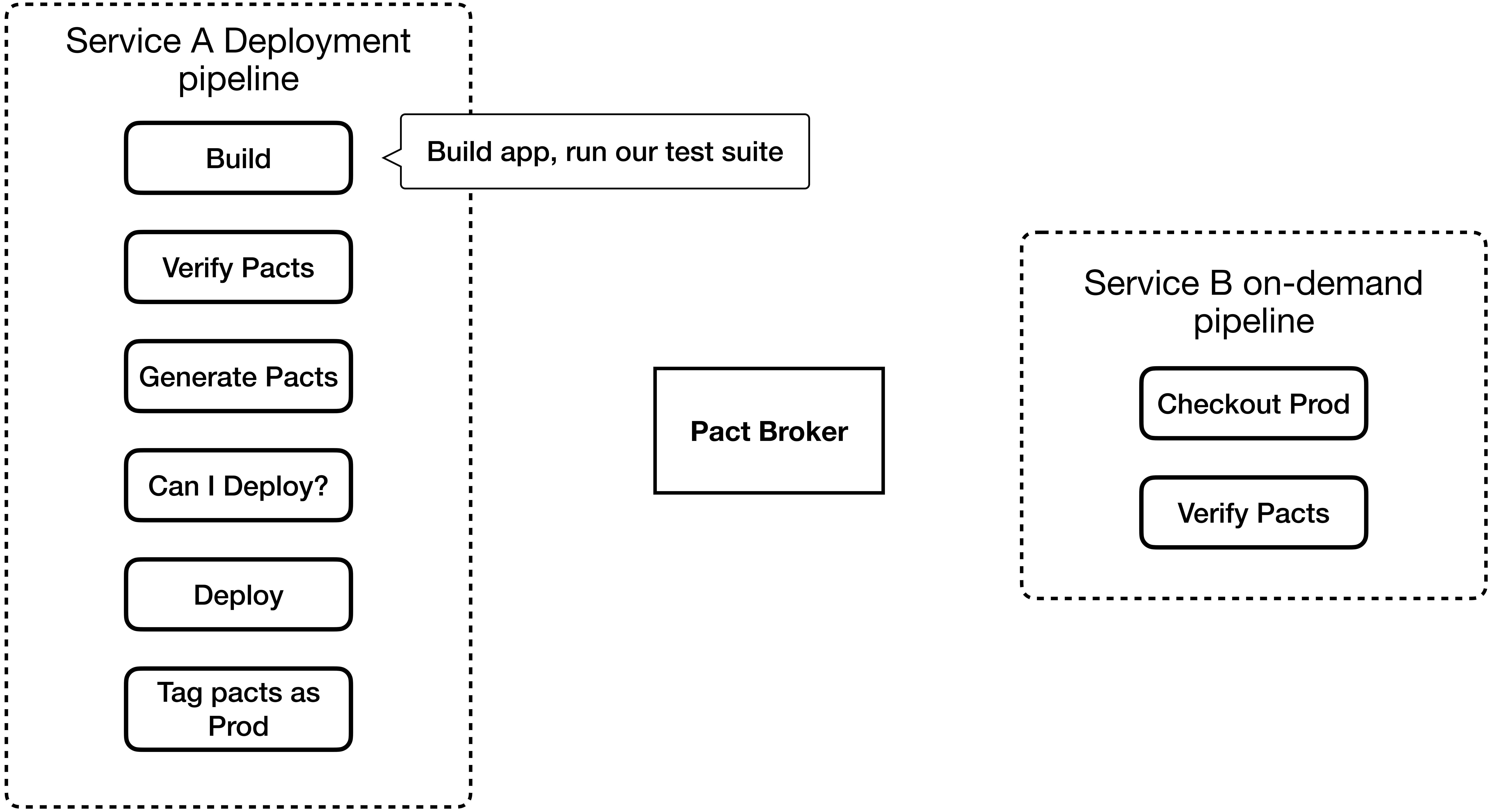
PACT

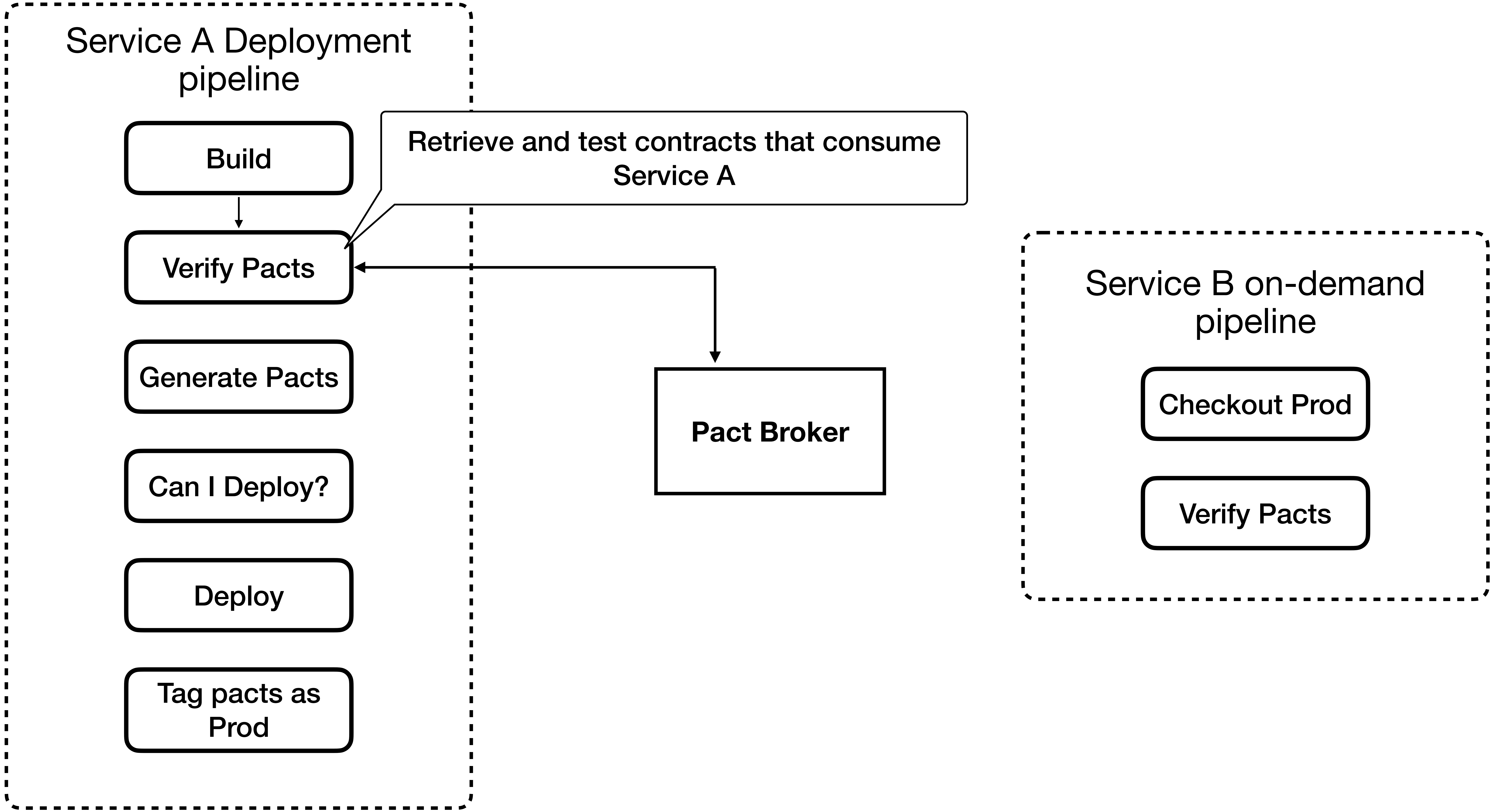
Pact Terminology 101

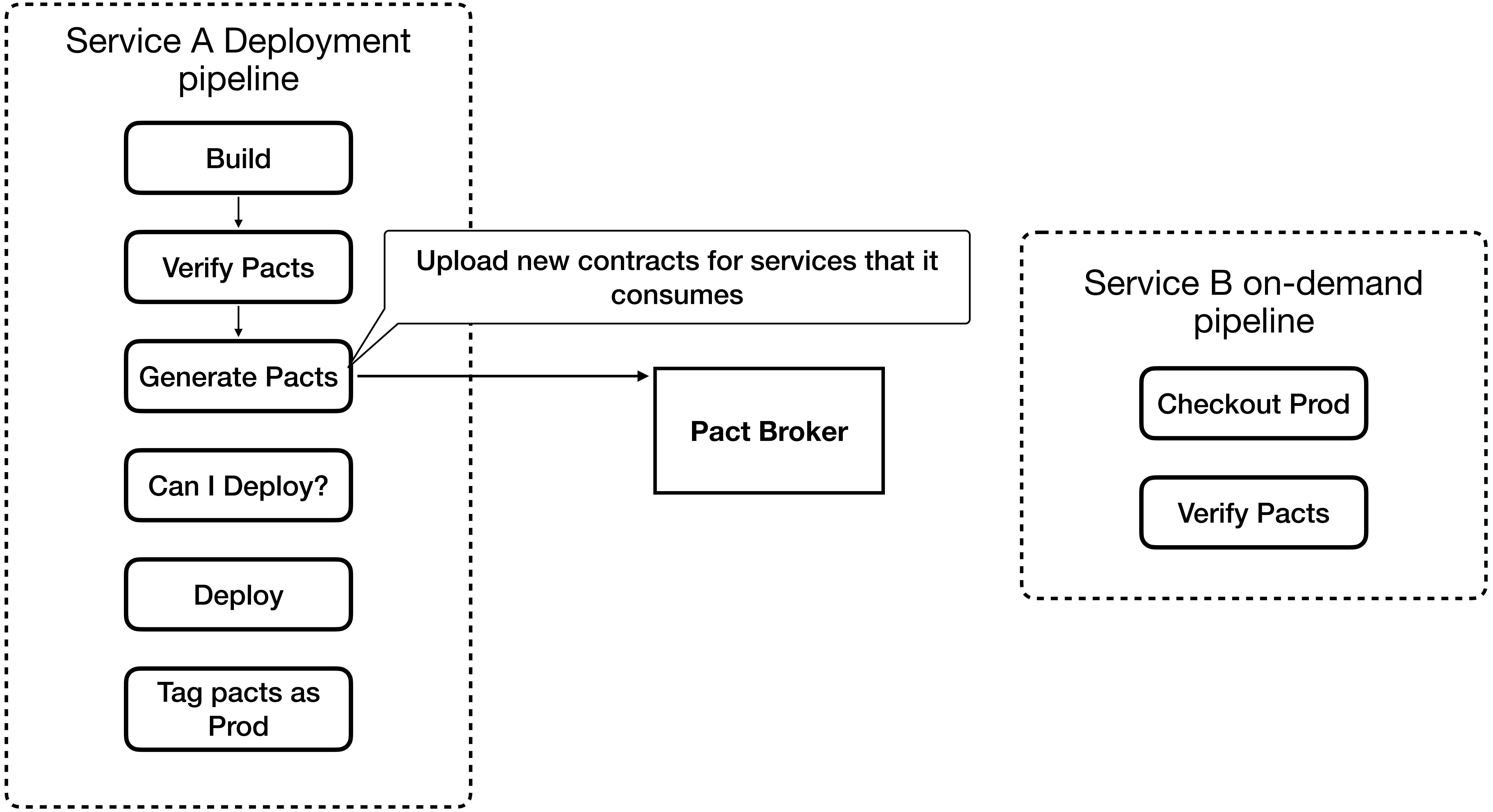
- Contract - shared understanding between a consumer and provider
- Consumer - a client that wants to receive some data (for example, web UI)
- Provider - provides the data that a client needs (for example, an API)
- Pact Broker - repository for contracts, manages versioning, web hook integration
- Consumer tests - tests against our client for correct request (unit test)
- Provider tests - retrieve contracts from Broker and test them against a real API for correct response (controller layer only to test HTTP/validation)

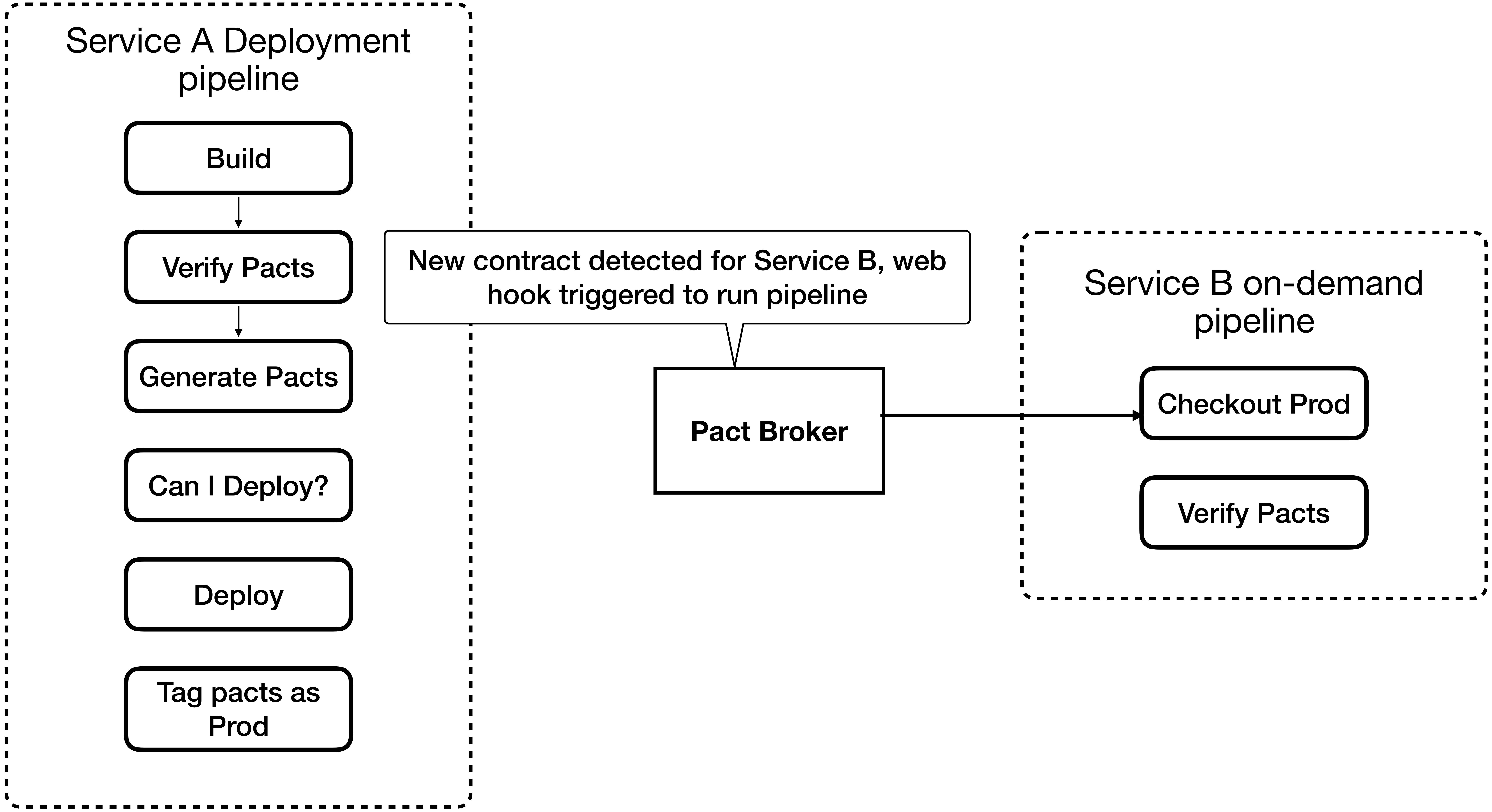
How does this help with microservices deployment?

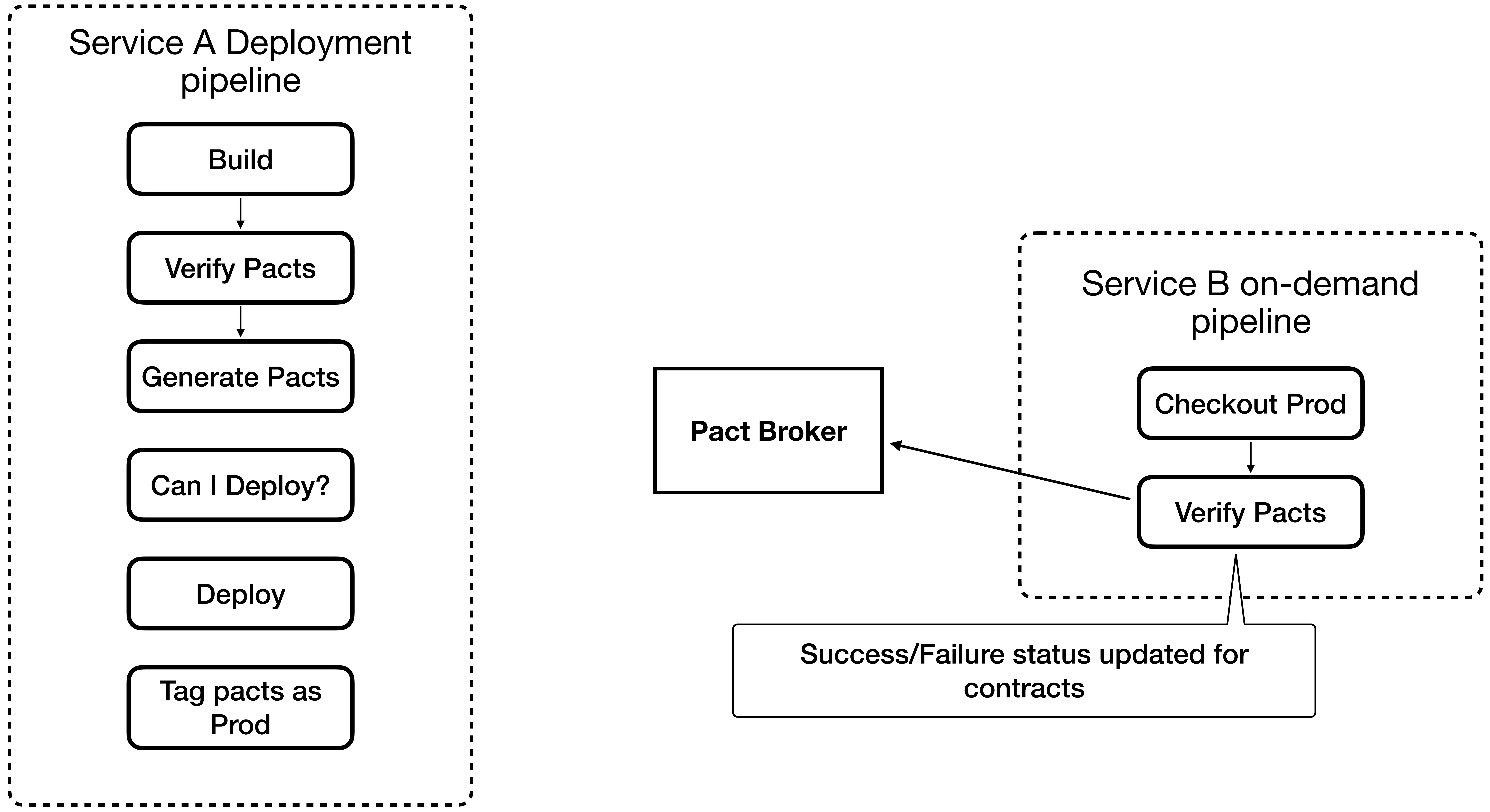

Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts

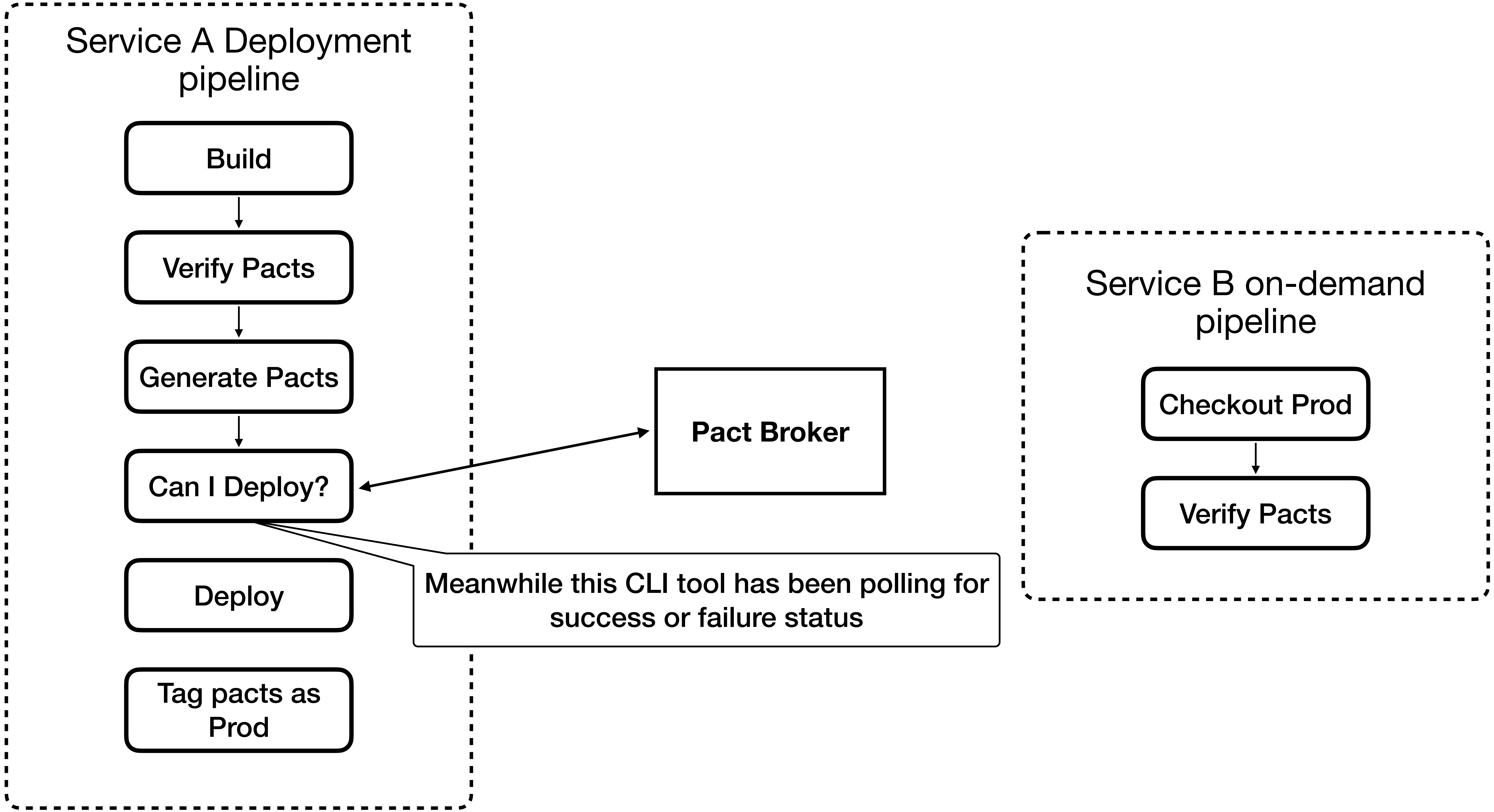

Service A Deployment pipeline
Build
Build app, run our test suite
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts

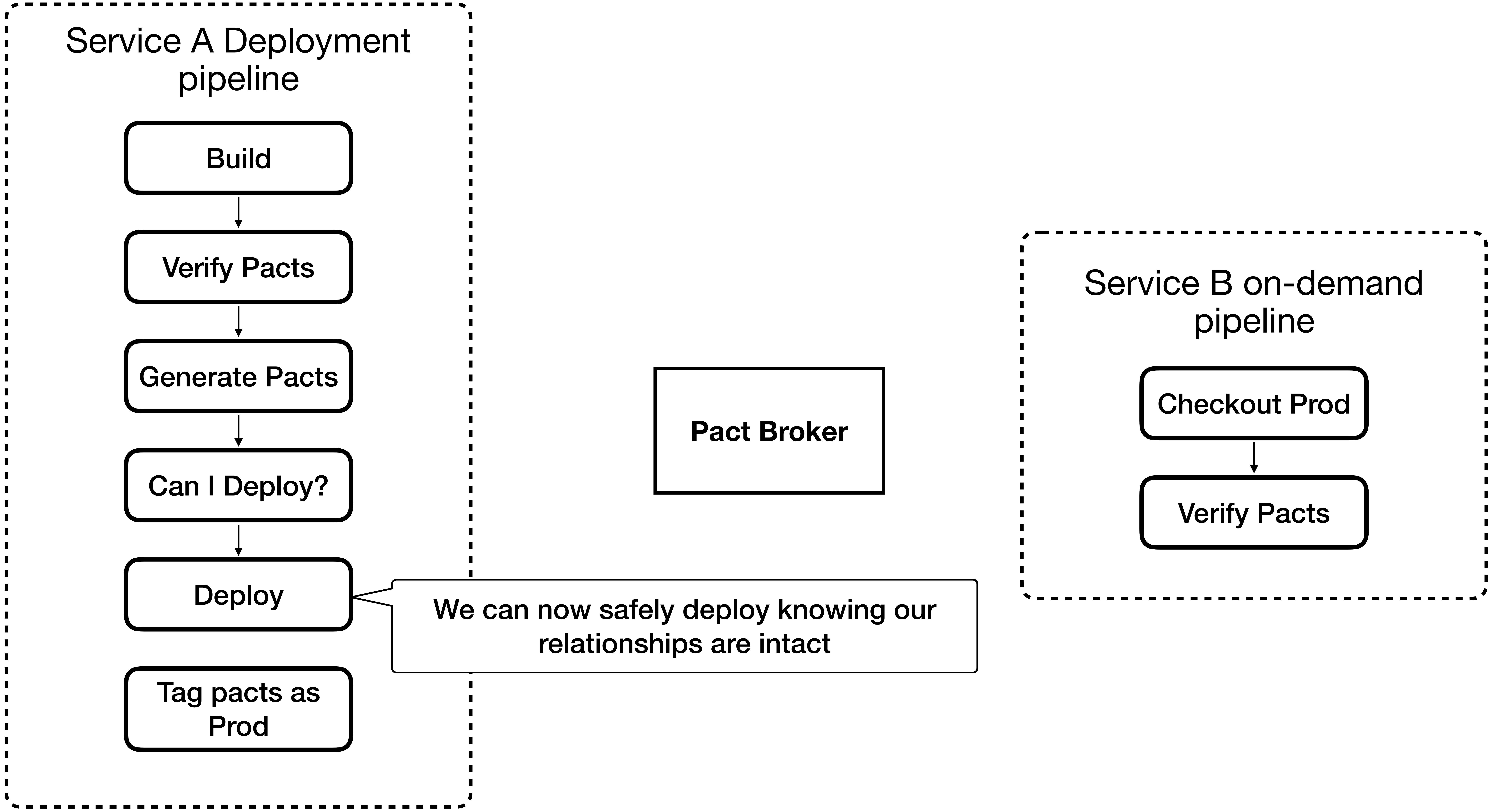

Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Retrieve and test contracts that consume Service A
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts

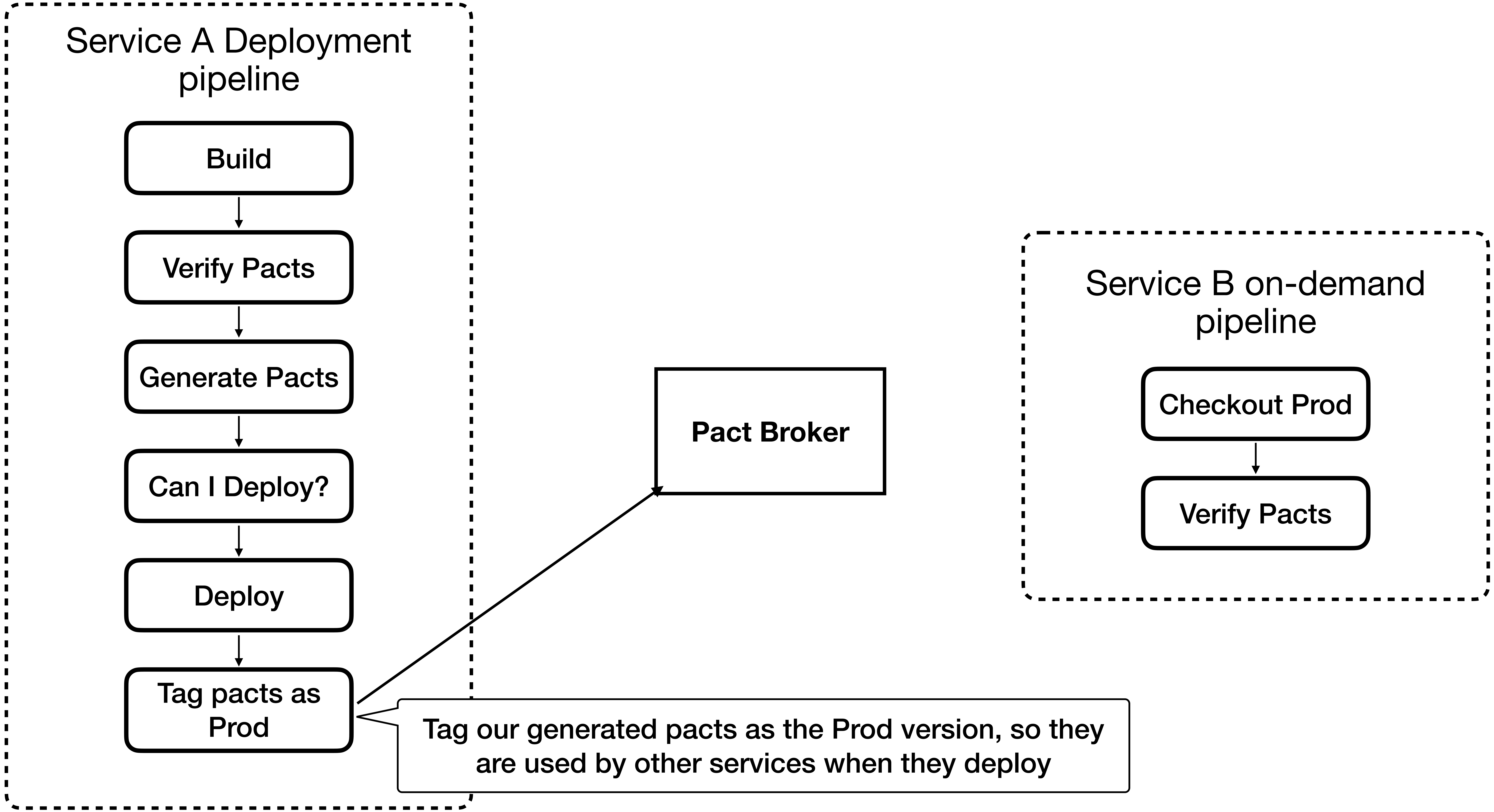

Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Upload new contracts for services that it consumes
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts


Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
New contract detected for Service B, web hook triggered to run pipeline
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts


Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts
Success/Failure status updated for contracts


Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Pact Broker
Meanwhile this CLI tool has been polling for success or failure status
Service B on-demand pipeline
Checkout Prod
Verify Pacts


Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts
We can now safely deploy knowing our relationships are intact


Service A Deployment pipeline
Build
Verify Pacts
Generate Pacts
Can I Deploy?
Deploy
Tag pacts as Prod
Pact Broker
Service B on-demand pipeline
Checkout Prod
Verify Pacts
Tag our generated pacts as the Prod version, so they are used by other services when they deploy

What have we gained?

What have we gained?

- Fast feedback - find deployment issues before they happen
- Act as a barrier to deployment (prevents breaking changes from being introduced)
- Issues can be debugged locally
- Replaces expensive, brittle integration tests with fast, stable unit and web layer tests (pushing tests down the test pyramid)
- Provides awareness of who consumes your API - Pact allows you to view dependency graphs

2. Build resilient and fault tolerant systems

Resiliency

- Microservices should be able to deployed individually, and not rely heavily on external services.
- We can build resilience and fault tolerance into our services to diagnose errors, handle errors gracefully, promote self healing, and ultimately achieve a higher level of success

Resilience4j

HYSTRIX
DEFEND YOUR APP

Resilience and Fault Tolerance patterns

- **Circuit breaker:**
  - When a system is seriously struggling, failing fast is better than making clients wait.
  - Example use case - if you know a service is down, fail fast and prevent redundant calls
- **Retrying:**
  - Many faults are transient and may self-correct after a short delay.
  - Example use case - integrating with a flaky API
- **Rate limiting:**
  - Limit the rate of incoming requests.
  - Example use case - prevent potential overload attempts, offload requests to more available instances
- **Fallbacks:**
  - No matter what you do, things will still fail - plan for a back up option
  - Example use case - if an API call fails, use cached version until it comes back online

What have we gained?

What have we gained?

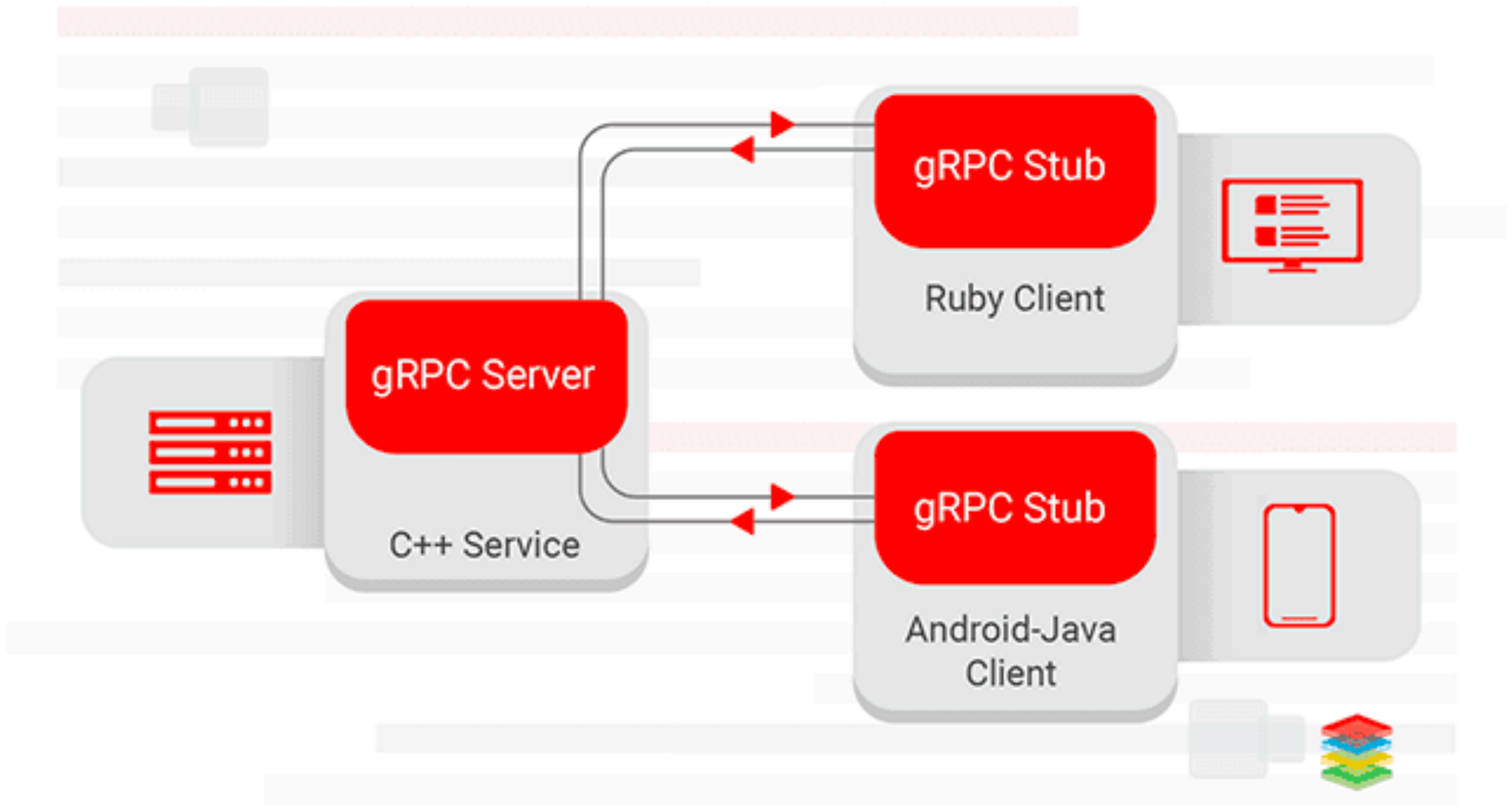
- Failures will always happen - we can handle them
- Including fault tolerance thinking leads to better design
- Enables applications to self heal - services can recover from failed states

3. Use the most efficient tools to mitigate network issues

gRPC

What is gRPC?

- In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure to execute in a different address space, which is coded as if it were a normal (local) procedure call.



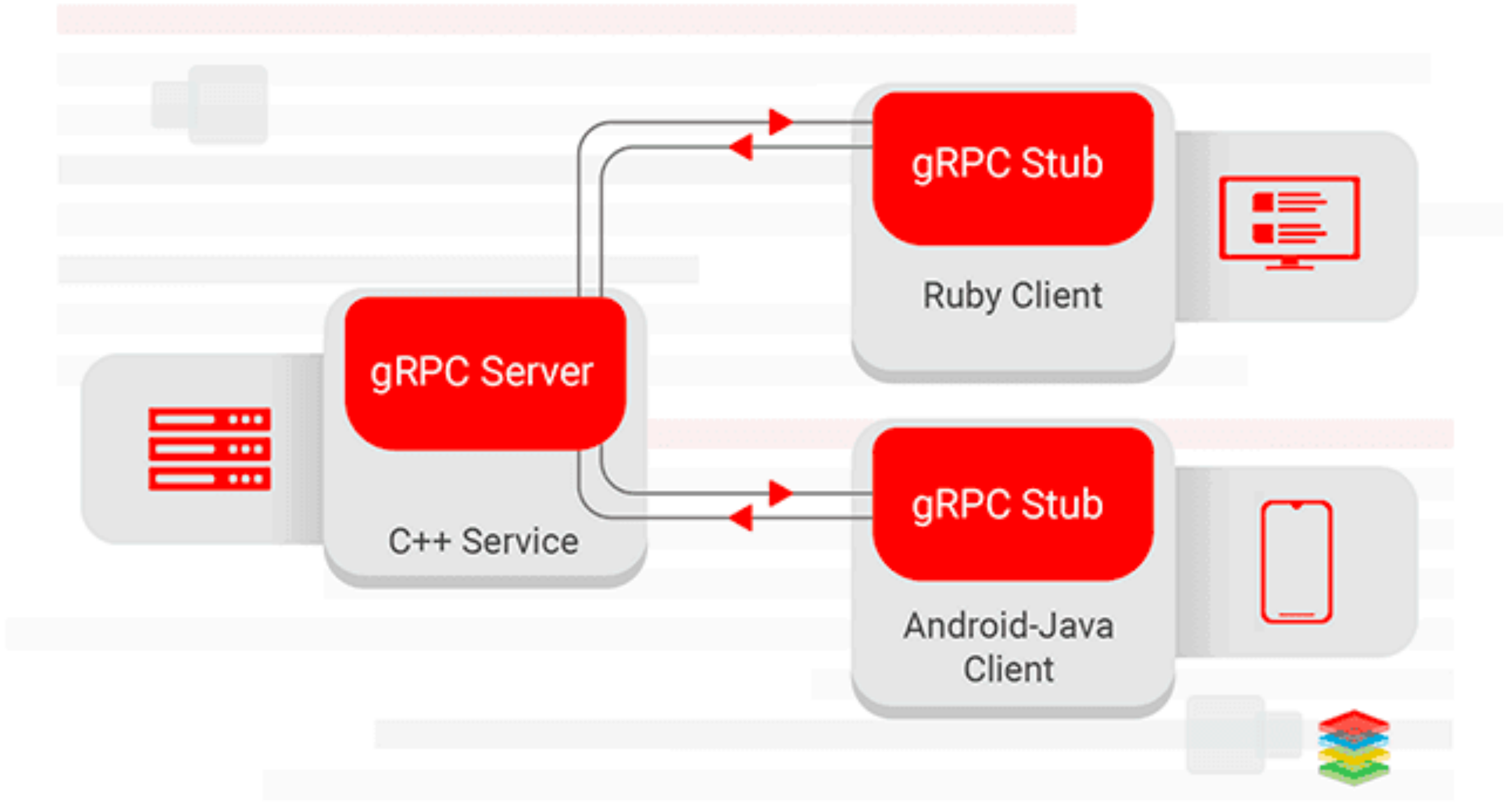
gRPC features

- gRPC is a communication protocol for services, built on HTTP/2. Unlike REST over HTTP/1, which is based on resources, gRPC is based on Service Definitions. You specify service definitions in a format called protocol buffers (“proto”), which can be serialized into an small binary format for transmission.

- With gRPC, you can generate boilerplate code from .proto files into multiple programming languages, making gRPC an ideal choice for polyglot microservices

- gRPC is designed for low latency and high throughput communication.

- gRPC is great for lightweight microservices where efficiency is critical.



What have we gained?

What have we gained?

- **Performance:**
  - gRPC messages are serialised using Protobuf, an efficient binary message format. Protobuf serializes very quickly on the server and client. Protobuf serialization results in small message payloads, important in limited bandwidth scenarios like mobile apps.
  - gRPC is designed for HTTP/2, a major revision of HTTP that provides significant performance benefits over HTTP 1.x
- **Code generation**
  - All gRPC frameworks provide first-class support for code generation. A core file to gRPC development is the .proto file, which defines the contract of gRPC services and messages. From this file gRPC frameworks will code generate a service base class, messages, and a complete client.
  - By sharing the .proto file between the server and client, messages and client code can be generated from end to end. Code generation of the client eliminates duplication of messages on the client and server, and creates a **strongly-typed client** for you. Not having to write a client saves significant development time in applications with many services.
- **Streaming**
  - HTTP/2 provides a foundation for long-lived, real-time communication streams. gRPC provides first-class support for streaming through HTTP/2. A gRPC service supports all streaming combinations (Unary, Server to client, Client to Server, **Bi-directional**)

Wrapping up this project..

Some extra things to consider

- **Shared configuration:**
    - Chances are, a lot of your microservices will require the same set up.
    - Sharing common configuration settings using a config server will allow configurations to be dynamically updated across all of your services.
    - Sharing container set up using custom Dockerfiles or Buildpacks will provide a consistent way of running your apps, using best practices
- **Fast development cycle:**
    - Microservices should enable developers to move quickly
    - If there is too much complexity, consider combining related services for the sake of simplicity

Congratulations, you have successfully moved into the world of microservices, and learned a bit about scalability, compatibility and fault tolerance… just want the Business™ wanted!