Reliably Releasing Software

Foundations of Software Engineering

Christopher S. Meiklejohn

Software Engineer, DoorDash

Adjunct Faculty, Carnegie Mellon University

Carnegie Mellon University

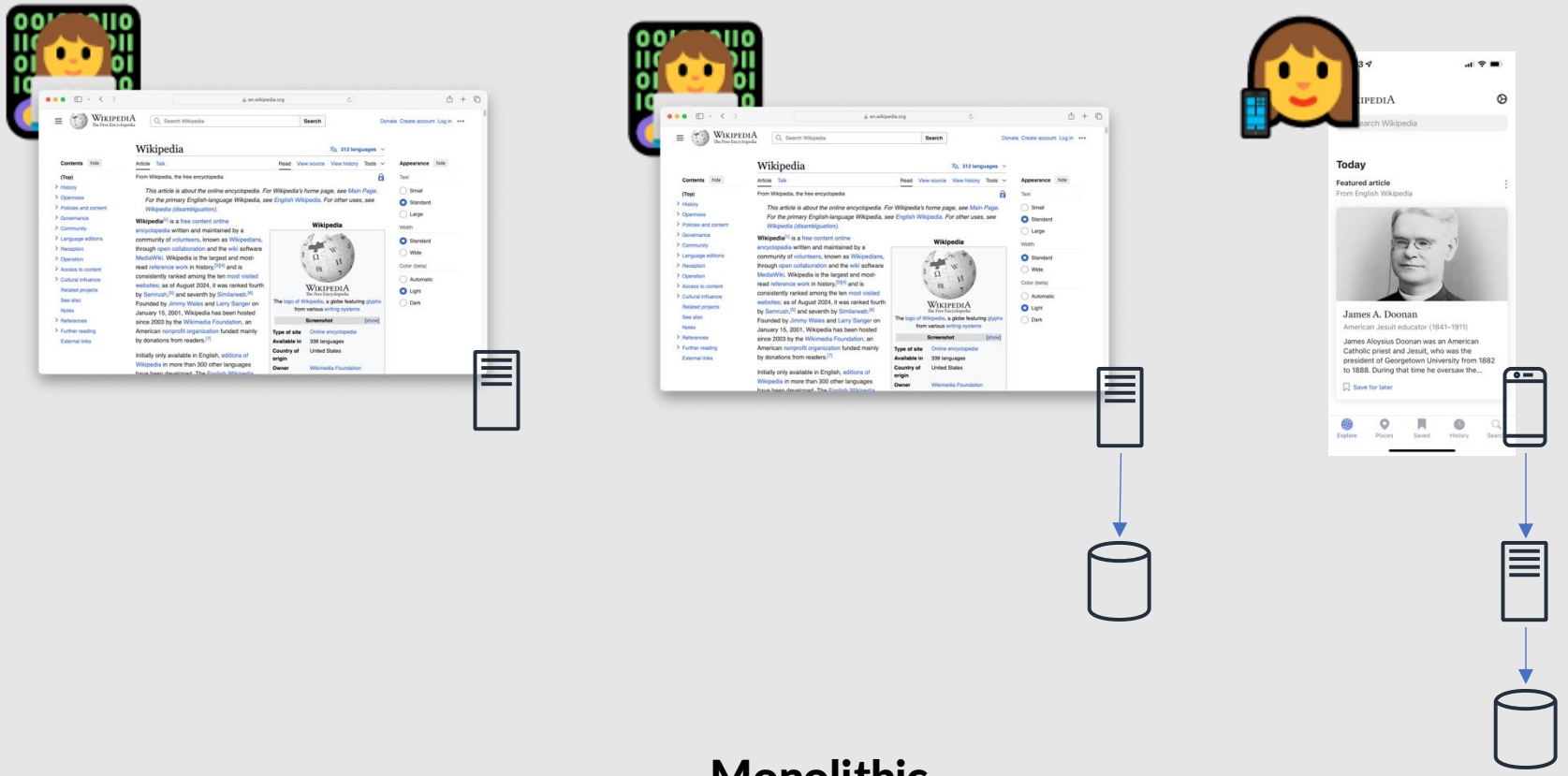
Let's Revisit

Identify the core challenges with modifying, testing, and deploying applications **safely.**

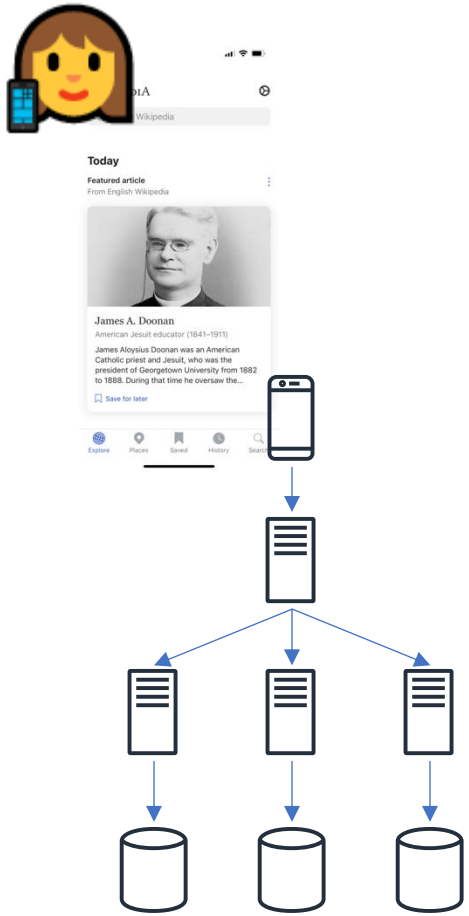
Describe and **differentiate** the possible techniques for ensuring **reliable** and **safe delivery of software at scale.**

Practice authoring a **safe rollout plan** for a new feature.

What About This Software?



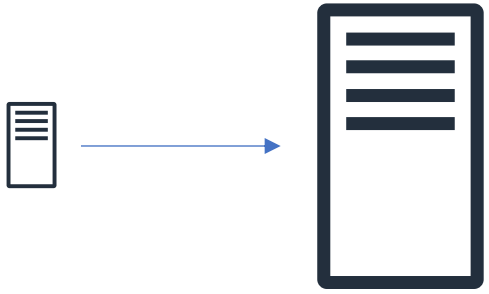
Microservice



Scaling and Deployments: Intertwined

Scaling

Vertical Scaling

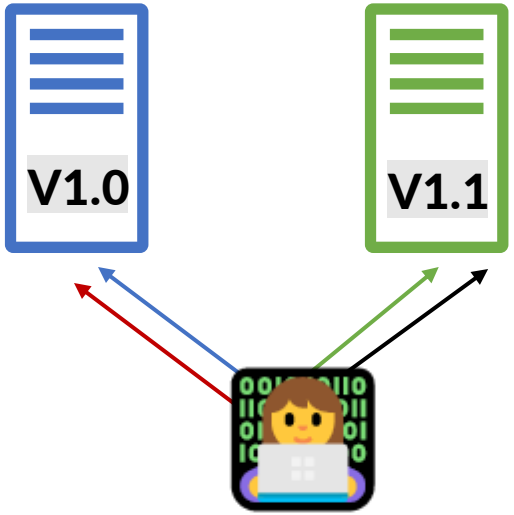


Horizontal Scaling

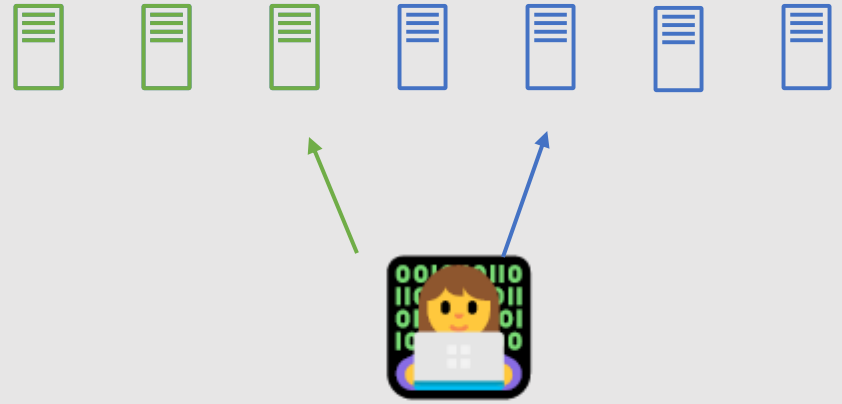
Red/Black: switch
Blue/Green: incremental traffic

Red/Black or Blue/Green

Deployment



Rolling Upgrade



Dark Launch

Solution: **Dark Launch**

Rollout with Features Dark
Perform rollout of code at the "same" existing version with all new features turned "off" – no-op rollout.

Incremental Ramp of Flag
Incrementally enable feature to users based on percentage and roll out to employee (or other limited cohort first) for early detection *(i.e., dogfooding.)*

Rollback: First Response
Ensure that code can be rolled back immediately on the first indication of issue.

Rolling Upgrade with Dark Feature

Incremental Feature Release

Remember to write tests with the feature flag = **false and true prior to rollout!**

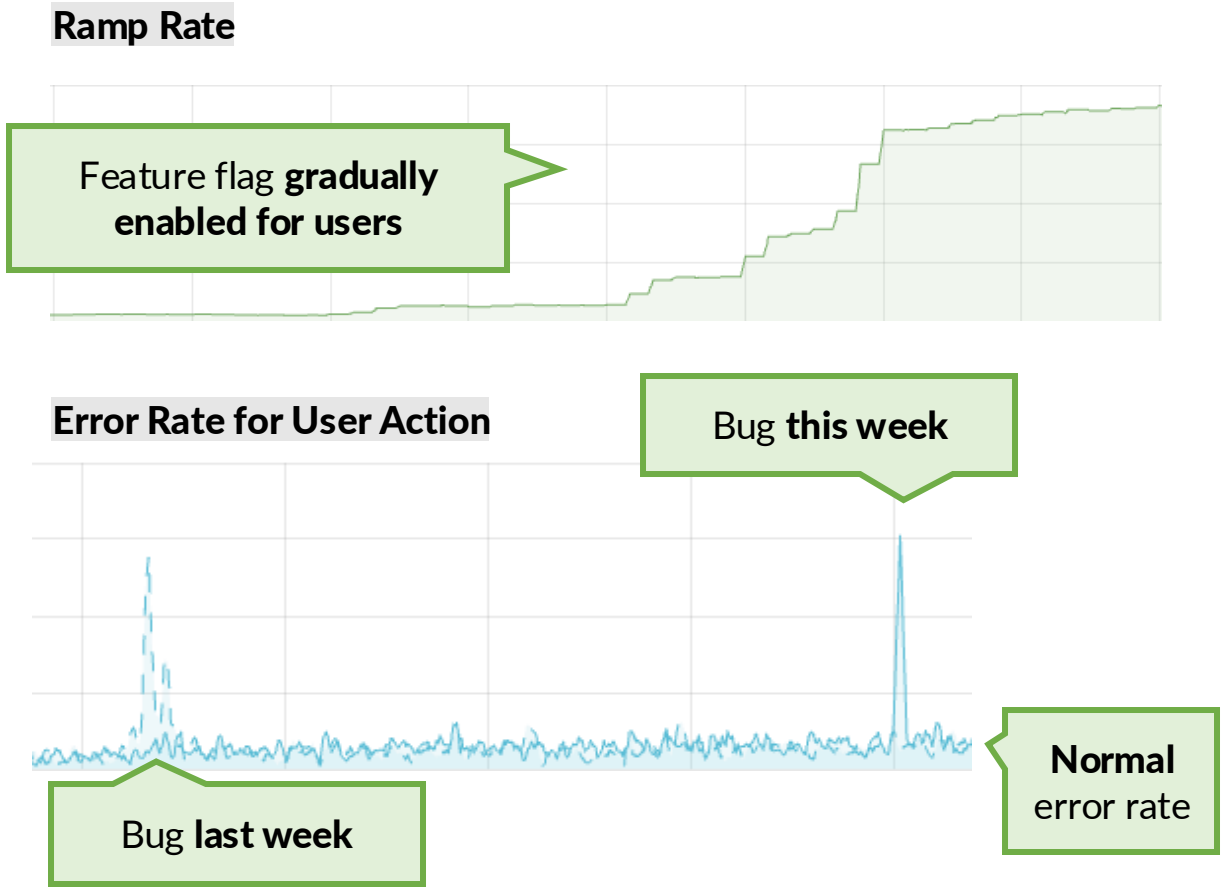
Dark Launch: Observability

How do you **identify a rollout problem?**

Hit Rate
Use metrics tracking new code execution to track introduction of new feature.

Error Rates
Use metrics tracking error rates and compare with week-over-week for derivations.

Remember: some errors may be normal depending on the metric.
Correlate them with the feature ramps.



What Are The Differences?

Location

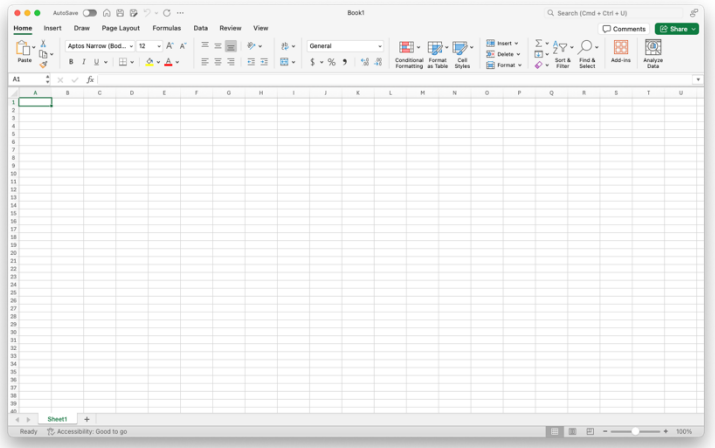
Servers, not Devices
Application runs on server and is **deployed to cloud.**
It's **not installed** on client's device.

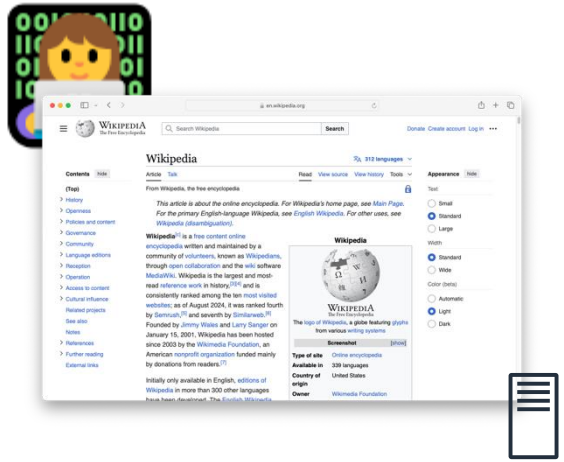
Scaling

"Scale out"
Scaling is achieved by increasing the server capacity, instead of installing the software on more clients.

Availability

"Always On"
Applications are upgrade in place, typically aiming for zero-downtime.





Bugs?

Rollouts Are Slow
Applications may have **thousands of server instances**, rollouts can take multiple hours.

Bugs Might Take a While To Surface
Error rate might be low, might take a while to detect, might be manually reported.

High Cost/Impact For Bugs
Every second of a bug may indicate possible user error. *(e.g., can't request a ride)*

Can't Immediately Rollback
Not enough capacity to immediately rollback *(i.e., blue nodes)* and deployment of old code is as slow as the new code.

Active Learning: Metrics

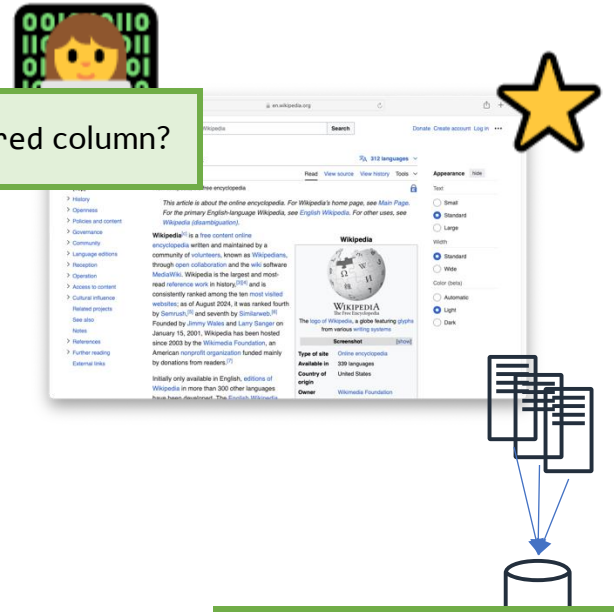
You are developing a **ride sharing application** and you're launching a **feature to allow users to request priority rides.**

Partner up with you neighbor and **answer the following:**

1.	**Define a metric that let's you track the feature rollout.**
	Rate of users who are eligible to see the priority option and then see the priority option.
2.	**Define a metric that let's you track successful usage of the feature.**
	Rate of users who see the priority option and receive a priority ride.
3.	**Define metric(s) that let's you track error rate of the option.**
	Rate of users who see priority option, select it, but do not receive a priority ride when available.
	Rate of users who should see priority option, but do not see the priority option.

Databases: What's Hard About This?

We have one database schema, **how do we change it?**
(recall: we have to add a new field called `starred`)



What type is the `starred` column?

No Rolling Upgrades
Can't synchronize rolling upgrade between app + database, no rolling upgrade for DB, even schema changes in distributed databases are atomic across nodes.

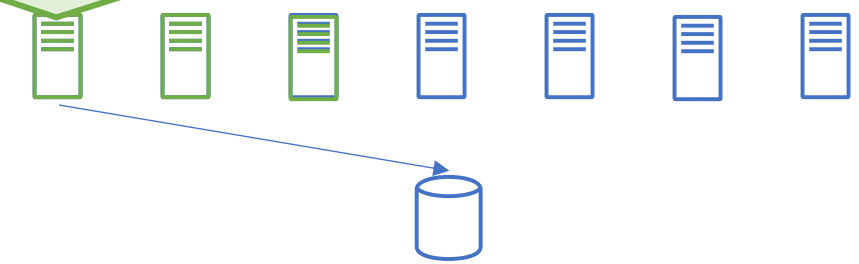
In short: changes are **atomic.**

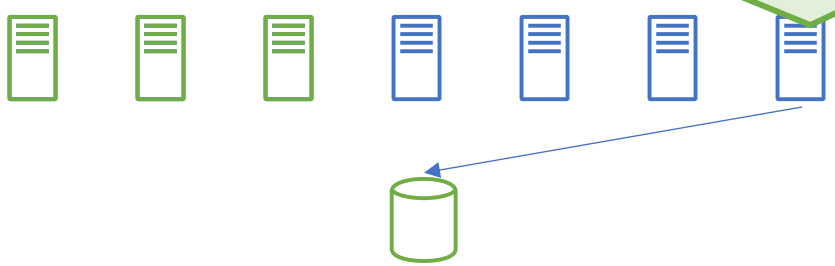
What type of problems does a **rolling upgrade of our app code** introduce if our DB change takes effect immediately?

New version might be **incompatible with old DB**
(i.e., access starred before there.)

Problems During Rolling Upgrade/Release

Old version might be **incompatible with new DB version.**
What scenarios might this be?





What About This?

Modifications to DB + App + Client
Many times you will have to modify the database with the application **and the mobile client** for new features.

Release Coordination
Can't synchronize updates: **mobile application modifications must be done ahead of time and submitted to the App Store/Google Play.**

Data Interchange
Backwards compatible message formats must be used and code must be able to **handle feature being absent/present.**

RPCs and Message Formats

Same problem as the database, **just with message formats and APIs, instead of the schema:**

1.	Data interchange **must be backwards compatible format**
	JSON, depending on the serializer and data mapping layer.
	Google's GRPC is **natively backwards compatible when adding new fields.**
2.	APIs must be **rolled out prior to mobile app that consumes them.**

Two new problem(s): **version longevity and forced upgrades:**

1.	One you make an API and it's used, **you own it for life as users may choose not to upgrade.**
2.	Backwards compatibility may have to be across **several versions.**

Key Takeaways: Backwards Compatibility

We haven't even talked about **microservices** yet!

You're (almost always) developing a **distributed system** even with a monolithic architecture.
(i.e., most monolithic applications use a database and have an associated mobile application.)

Therefore, the key to **safely rolling out changes** is **backwards compatibility.**

1.	Backwards compatible **database changes.**
2.	Backwards compatible **message and data formats** for data interchange.

Key Takeaways: Rollouts and Rollbacks

Backwards compatibility with **controlled rollouts** where **rollback is always possible.**

1.	Release features **dark, using feature flags or other mechanisms.**
2.	**Controlled rollouts** over time to mitigate risk by **gradually introducing changes.**
3.	At every step, **ensure you have the ability to rollback.**
4.	Have a **rollout plan and runbook for every step.**

Key Takeaways: Backwards Compatability

How do you ensure that code is **backwards compatible:**

1.	Test existing features with **feature flag = off, to ensure no regressions and no-op/dark rollout.**
2.	Test existing features with **feature flag = on, to ensure no regressions in existing behavior.**
3.	Test new features with **feature flag = on, to exercise dark launched code.**
4.	Testing should **include legacy data formats.**
5.	**Cleanup tests** after rollout.

Be a **good citizen!**

Key Takeaways: Deprecation

When you must make a **backwards incompatible change:**

1.	Use **tiered deprecation** where possible.
2.	At **minimum 3 rollout events**: add v1/v2 compatibility and enable v2, disable v1, cleanup.
3.	Some APIs have to be supported “for life”, **if they are exposed to clients and end users.**

Not only clients,
but also APIs.

Recall: Rollout Plan

What should be included in a great **rollout plan:**

0. **Testing.**
1. **Steps** to take in rolling out your change in sequence.
    a. **Backwards compatible** changes for new features, launched dark.
    b. **Tiered deprecation, 3-rollout strategy** for breaking changes **only if necessary**.
2. **Metrics** to monitor at every single step along the way.
    a. **Positive** *(e.g., feature hit, feature candidate for success, feature success)*
    b. **Negative** *(e.g., feature selected, didn't get, feature not present as option)*
3. **Rollback strategy** at every in the plan.
    a. Need to be able to **revert** every step if something goes wrong.

Microservice Applications

Microservice architecture is an architectural style where applications are constructed from services that communicate over the network using RPC and are developed, scaled and deployed independently.

NETFLIX

1,000 services
(2021)

UBER

2,200 services
>120 for getting ride
(2016)

DOORDASH

500 services
>100 involved in core flow
(2024)

Microservice applications are the **most common and complex** type of distributed application being built today.

Twitter (2017) operates a > 10k node distributed Hadoop cluster.
However, **most nodes have the same behavior, running the exact same code.**

DoorDash (2024) operates 500 microservices.
Each service provides **different functionality, has a different API, and is deployed continuously.**

Microservices: Socio-Technical Problem

Microservice architectures solve a **socio-technical problem:**

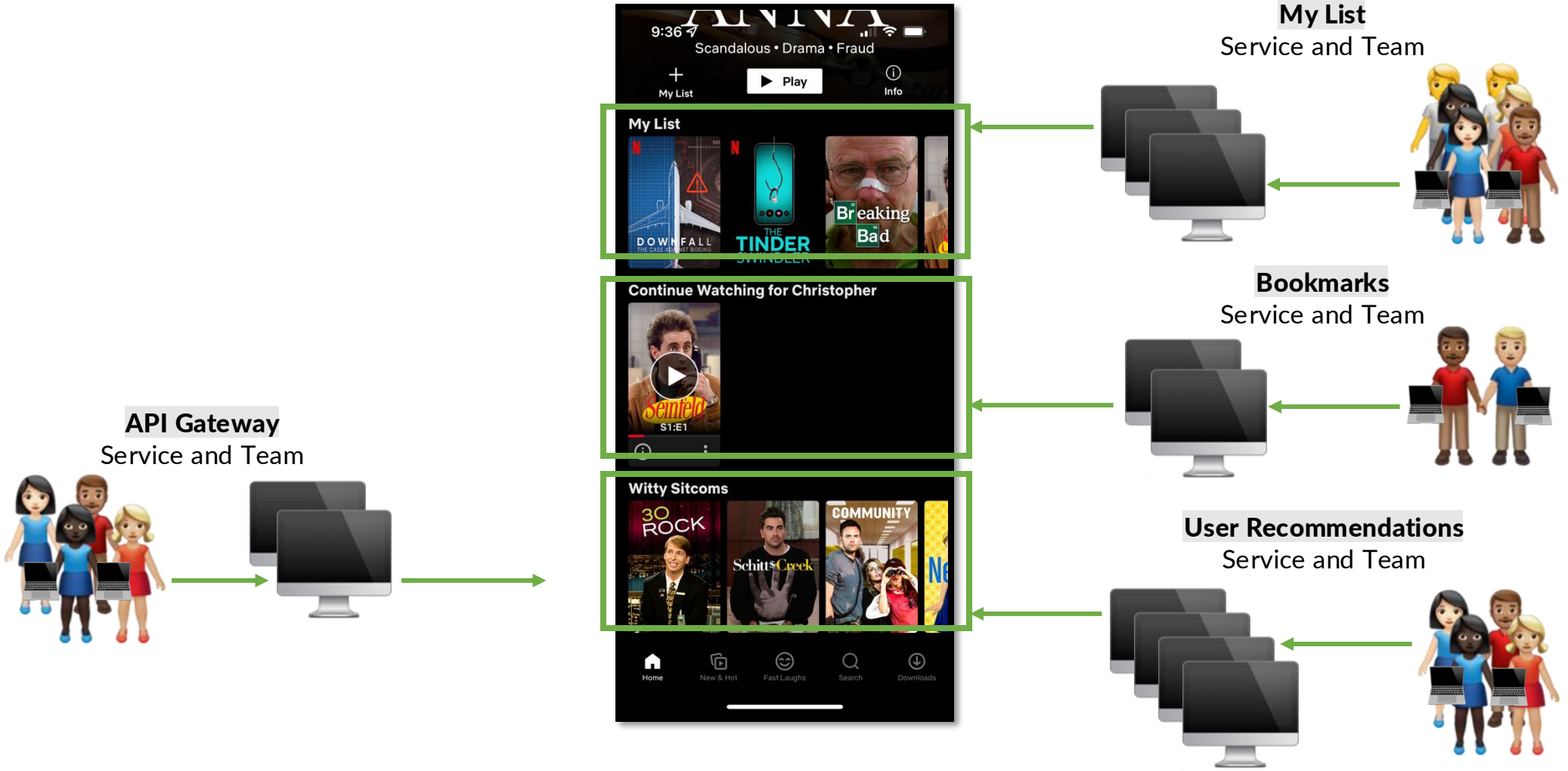
Technical solution to **support rapid feature development at scale** as an organization grows, that breaks down the application into components where no single engineer needs knowledge of the entire application to develop and deploy features.

We would not develop an application this way **unless it was absolutely necessary.**

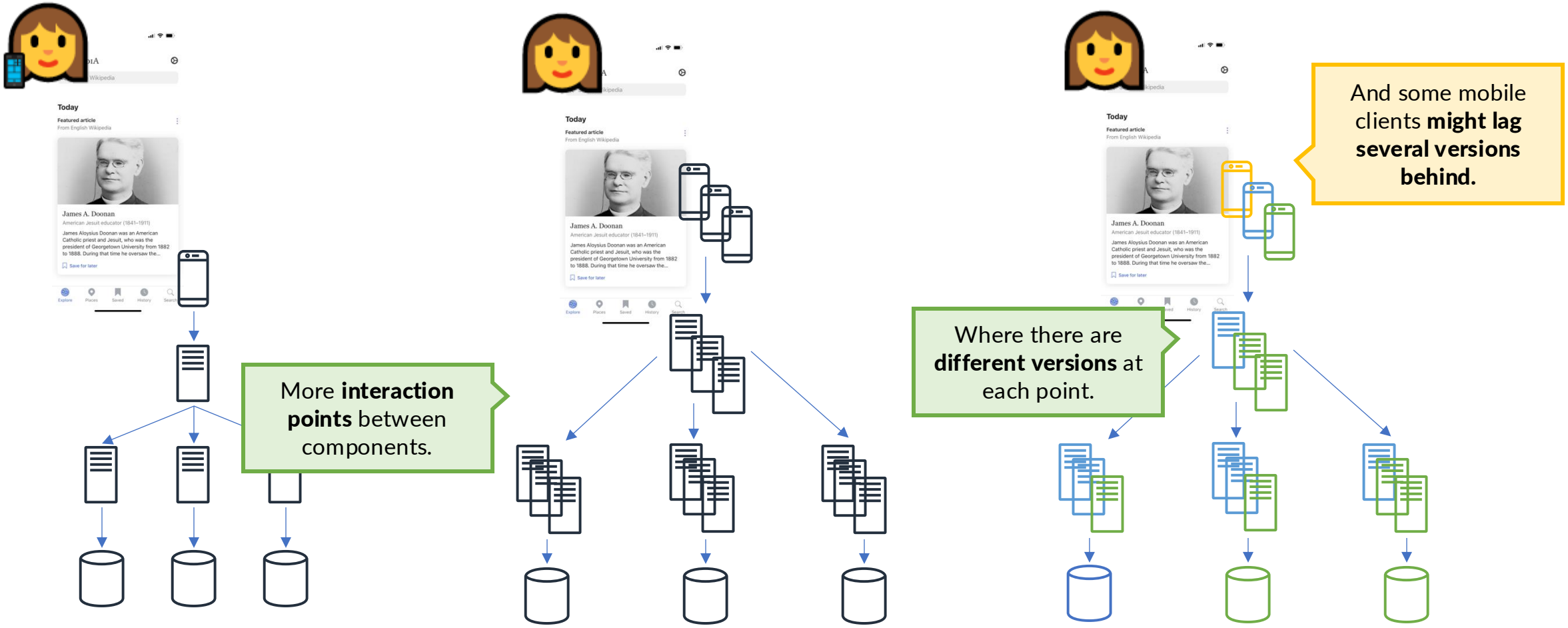
Technical solution splits code across multiple repositories (and languages) making **it harder to develop, test, analyze, and reason about the application.**
(e.g., IDE support, static and dynamic analysis tools, integration and functional testing, etc.)

Consequence: forces all developers **to become distributed systems engineers.**

Netflix: Microservice Architecture



Revisiting: Wikipedia



Microservices and Backwards Compatibility

Microservices **combine the problems of everything we've seen so far**, but also:

1. Introduce message/data interchange interaction points **between all services.**

Same problem as the **mobile client**: **backwards compatible messages and formats required** where the "downstream" service must provide an API before it can be consumed by an upstream.

Rate of adoption of new fields, downstream changes, by upstream out of your control: **still may need to own APIs indefinitely, or at least a long time.**

2. Increase **the testing burden.**

Anyone you call makes a change *(i.e., "downstream")*, **you have to run tests for all features that touch that with all combinations of feature flags (theirs, and yours.)**

Anyone who calls you *(i.e., "upstreams")* **must run tests for all features that touch that with all combinations of feature flags (theirs, and yours) whenever you make a change.**

Bad enough? This must be done **transitively for all services in the call chain.**

Testing: Knowing Your Service

First key to success: safe, dark rollouts with a rollout plan.

Second key to success: **know your own service behavior through testing:**

1. Don't overly focus on unit testing, **which typically mock too much behavior.**
2. Invest in **"functional service"** testing. *(i.e., treat your service as a function and test it's input and output)*
   a. **Test** each of your APIs given a request you would receive from "upstream" service.
   b. **Assert** that the intended response is returned by your service given the input.
   c. **Mock** "downstream" service calls using mocking framework with expected result based on the contract that they provide to you.
3. Test **variations of feature flags** to ensure proper coverage.
4. Ensure **negative cases are tested**: validation failures, errors from downstreams, etc.

What's my **risk** here?

This detects **regressions in your own service code.**

Mock Drift

Mocks can **drift** and are only good when they reflect the **actual service's behavior** you're calling:

1. May get false positives where service works with mocked responses but **fails with real services.**
2. Run the *same* tests **without mocked responses, to verify behavior matches real behavior.** *(i.e., integration tests.)*
3. Real services responses **might change from test execution to test execution because any downstream services may employ their own feature flags** which may alter responses.

Detection of Mock Drift

Detecting mock drift can **reveal bugs** and **service behavior changes:**
(assuming good enough assertions)

Isn't 100% true, but is a **good litmus test for what's happening.**

1.	**Test passes, mock doesn't match:**
	Backwards **compatible** change is made to interaction. *(e.g., new fields in the response)*
2.	**Test fails, mock doesn't match:**
	Backwards **incompatible** change is made to interaction. *(e.g., field has value change)*

If your tests appear "flaky" it's because there's **something you're not controlling for** that's changing between executions:
often a feature flag that's partially ramped.

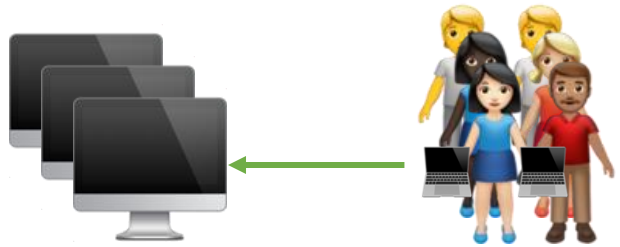
If all tests passed, no tests were added, **but mock drift detected:**

1.	Missing **necessary test coverage** for a new feature. *(e.g., field has no assertions)*
2.	Data is being **passed through you to an upstream caller and not used directly by your service.** *(e.g., field has no assertions)*

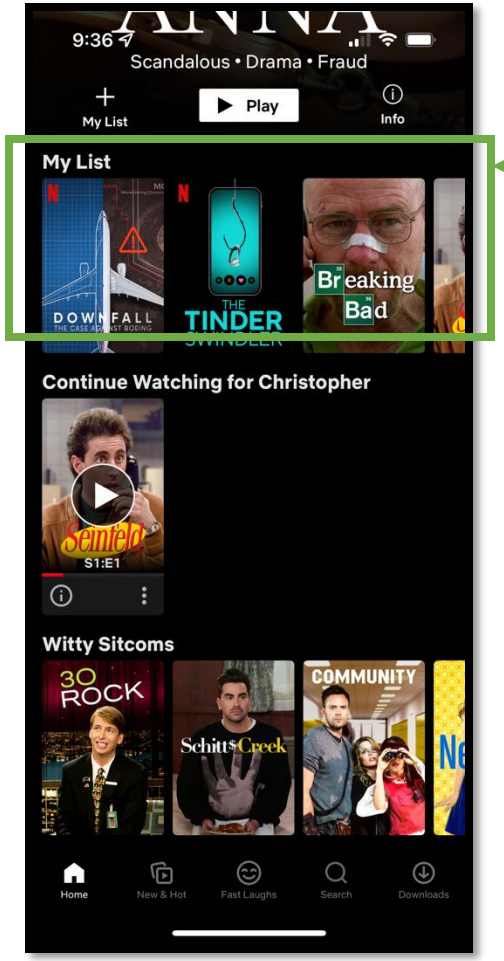
"Worst" changes to test **because it means that errors are not "encapsulated" to the calling service – error might surface in any "upstream" service.**

Netflix: Data Propagation

My List
Service and Team



If API Gateway just **passes data directly from My List to mobile client…**



…change in My List **could break mobile client and be undetected unless explicitly tested.**

API Gateway
Service and Team

By writing assertions for data you're passing upstream **it will allow you to preemptively detect issues that will eventually surface elsewhere.**

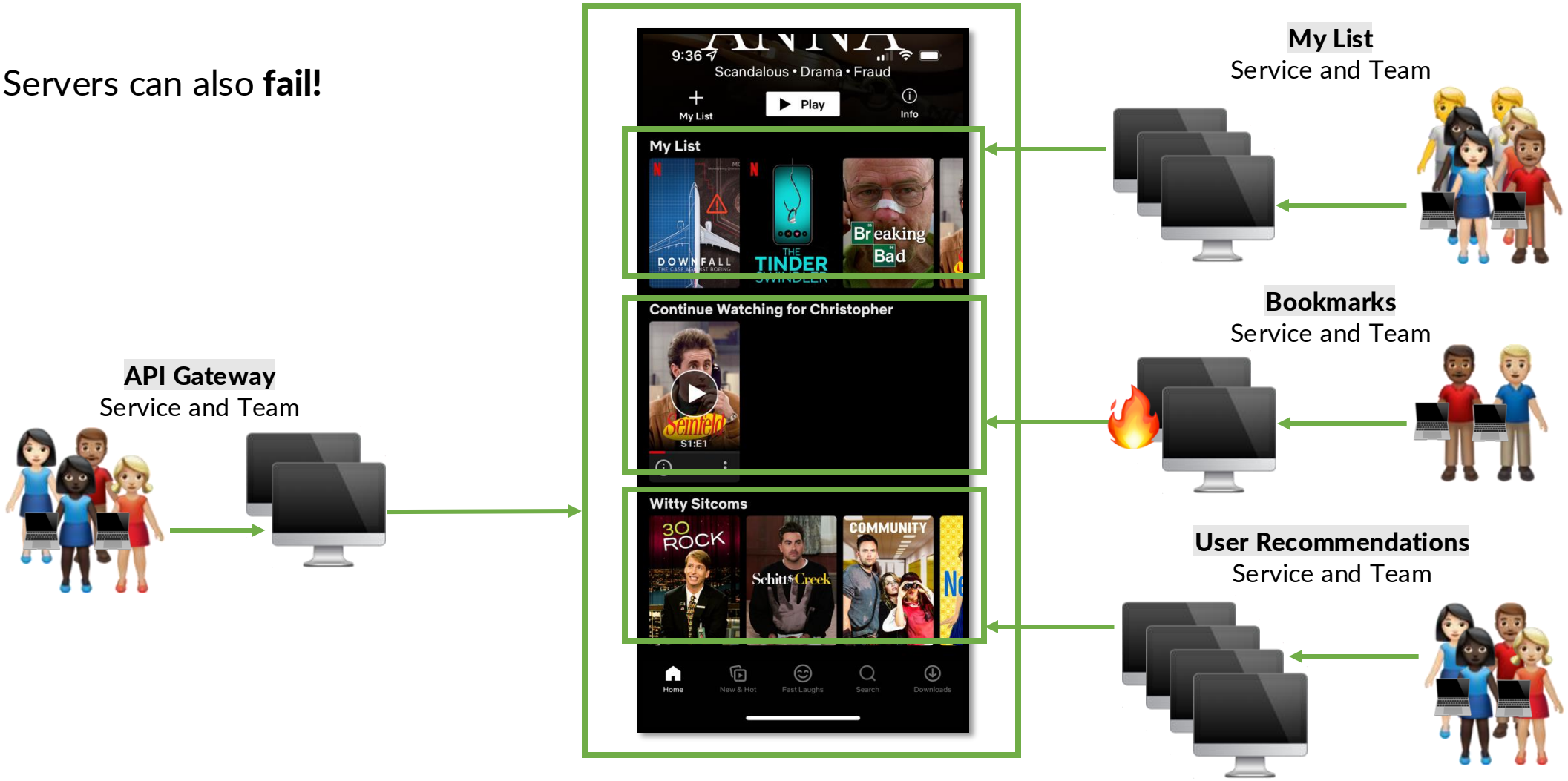
Testing Gotchas!

1.	**Be wary of manual testing.**
	Manually testing your feature (with the feature flag = true) **doesn't mean that all other features stay working when your feature is on.**
	Make sure you **hit the actual code paths where you made modifications**. This may require detailed logging or instrumentation to be sure.
2.	**Just because it worked in testing doesn't mean it will work in production.**
	The services you call, your "downstream" services, **can change between the time you tested and the time you actually deploy and turn your feature on.**
	Production environment **may not match testing environment.**
	If you can, test in production with just your user before rolling your feature flag out. Perform a final **"smoke test."**

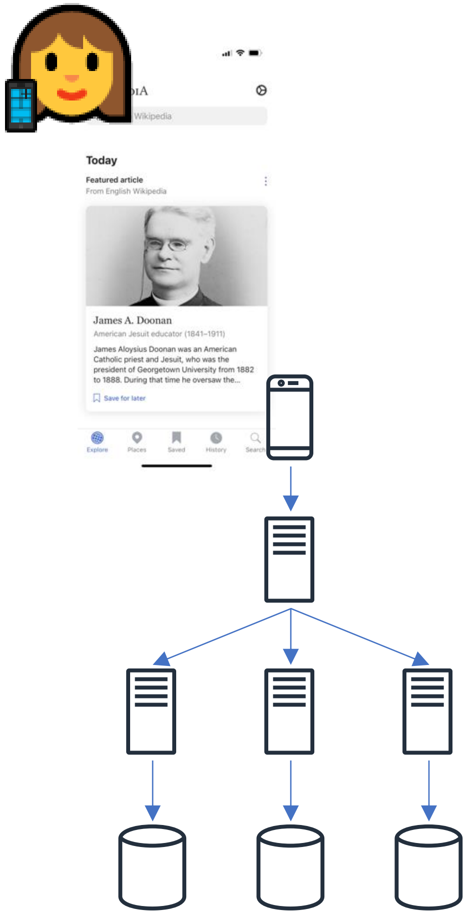
It's always better to be overly conservative, as
you'd rather find the bug and not hear it through revenue loss or a customer complaint.

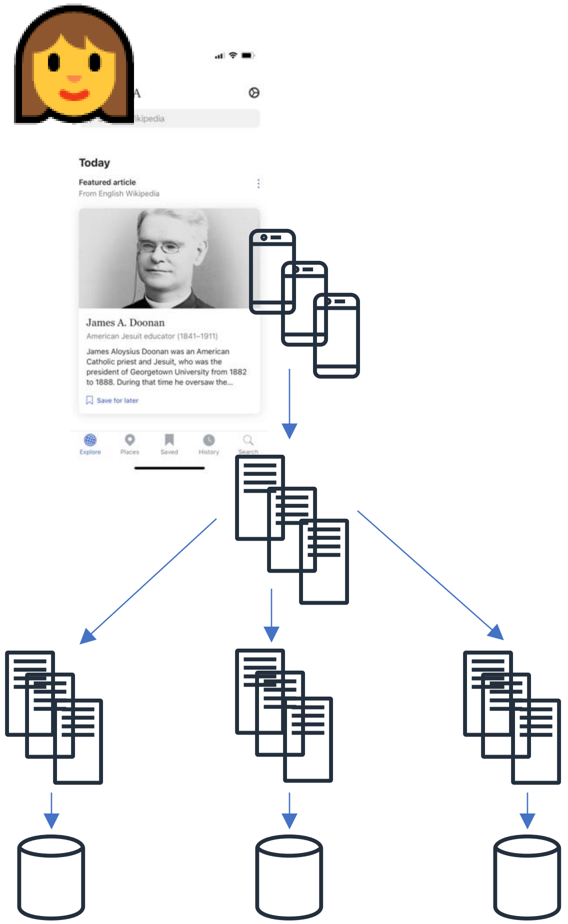
Just One More Thing

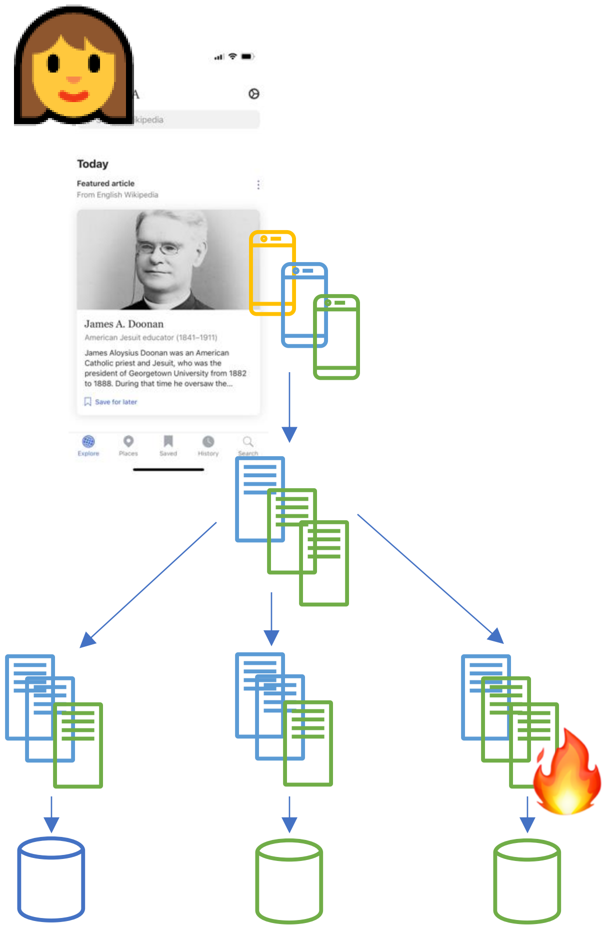
Servers can also **fail!**



Partial Failure







Partial Failure in Microservices: Different

…but, microservices are also susceptible to **partial failure:**

1. **Failed node causing connection errors.**
   Prior to removal by health check, application must still tolerate and respond to errors.

2. **Bad deployments.**
   Number of nodes return error responses *(e.g., 500 Internal Server Error)* before removal.

3. **Service failures only with certain arguments.**
   Service returns errors when provided with certain arguments by a caller only. *(e.g., NPE, etc.)*

4. **Dependencies of a given RPC method may be malfunctioning.**
   Direct dependencies of a service may slow down, timeout, or fail in other ways.

Challenges:

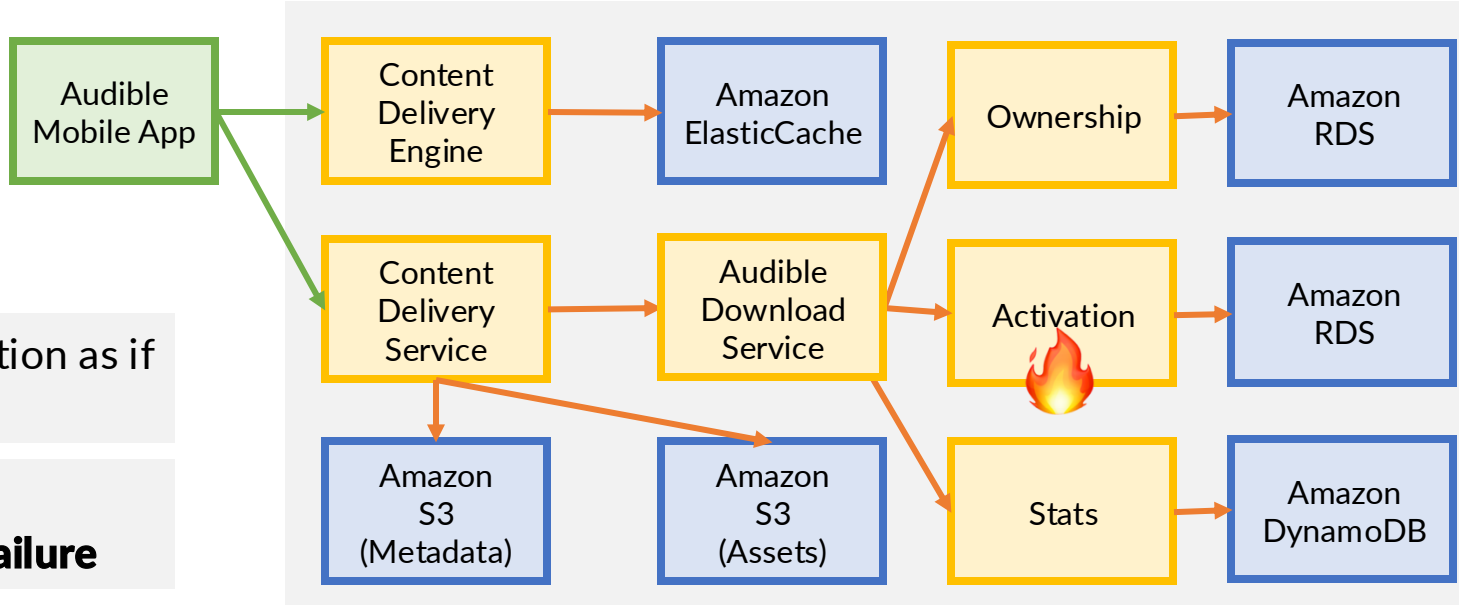
1. Developers **may not consider partial failure** unless they've (recently) encountered it before.
2. Success at scale **requires automation** in the local development environment and CI/CD pipelines.

Microservice Application: Audible

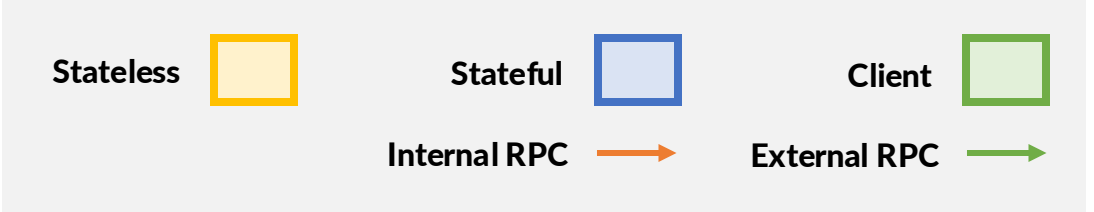
One solution to **partial failure:**

1. Build the microservice application as if it's a **monolithic application**
2. Fail the entire request **if any dependency returns a failure**

These are called **hard dependencies.**



Audible
Audiobook streaming service

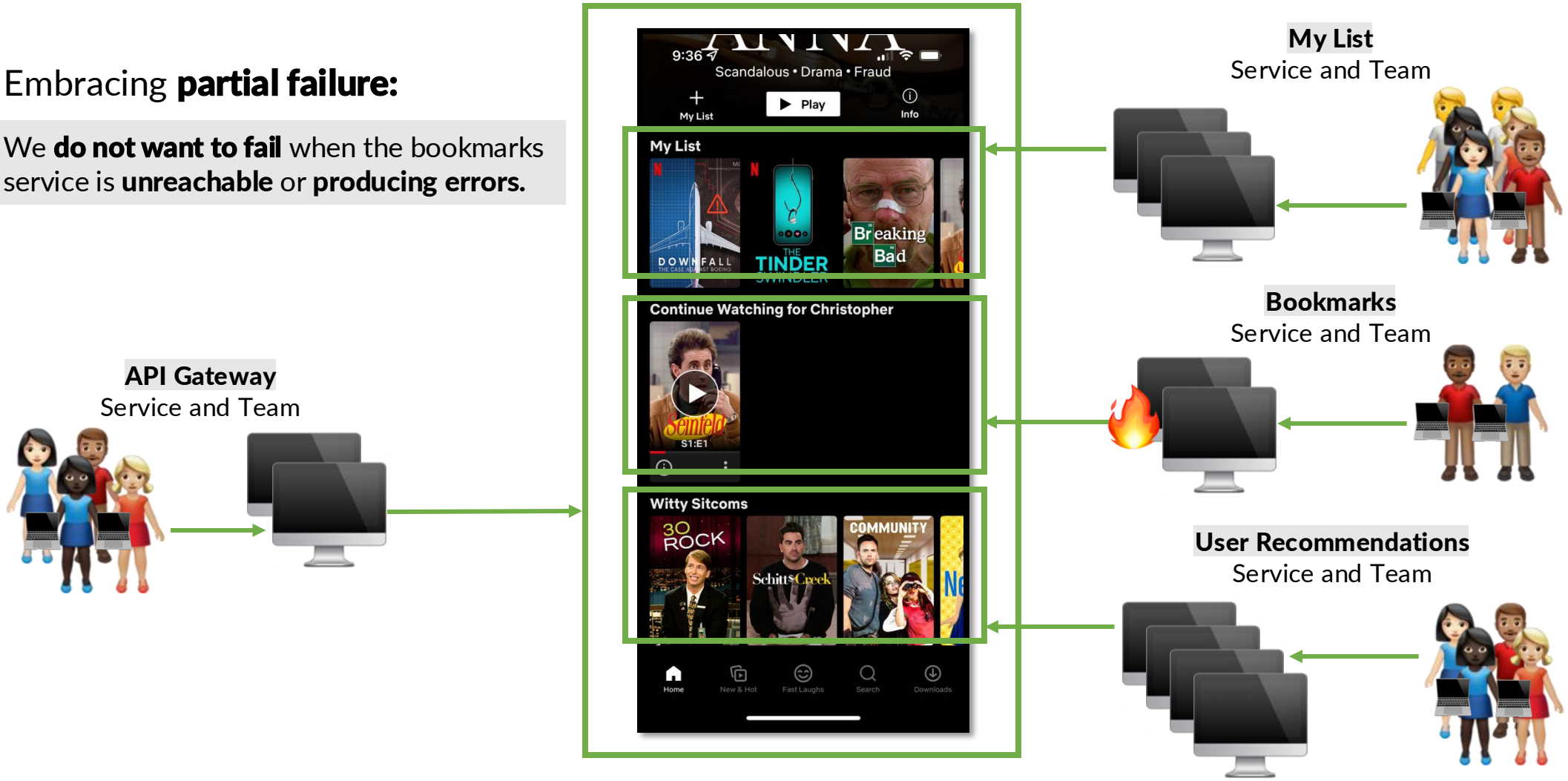


Alternatively,
should we **embrace failure?**

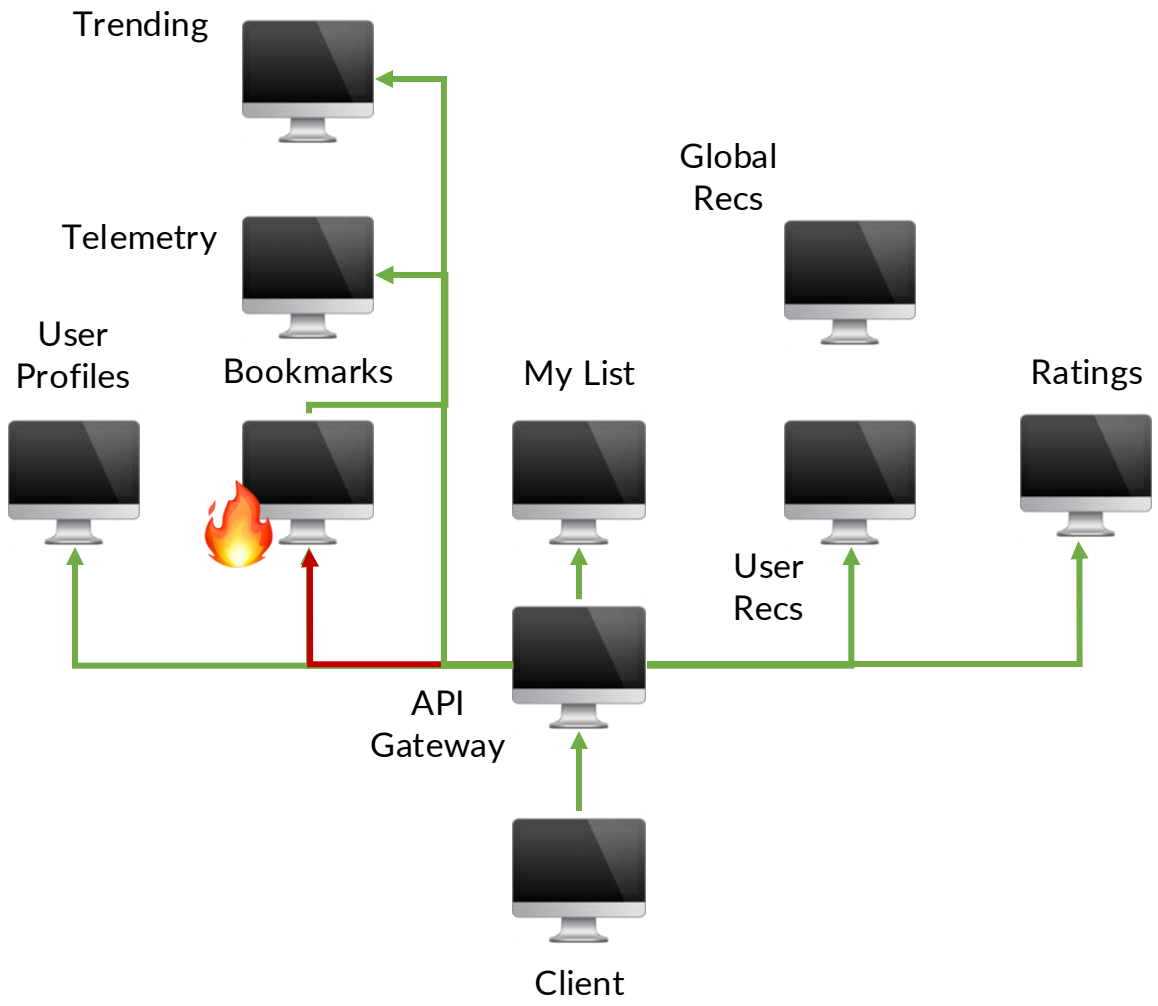
Microservice Application: Netflix

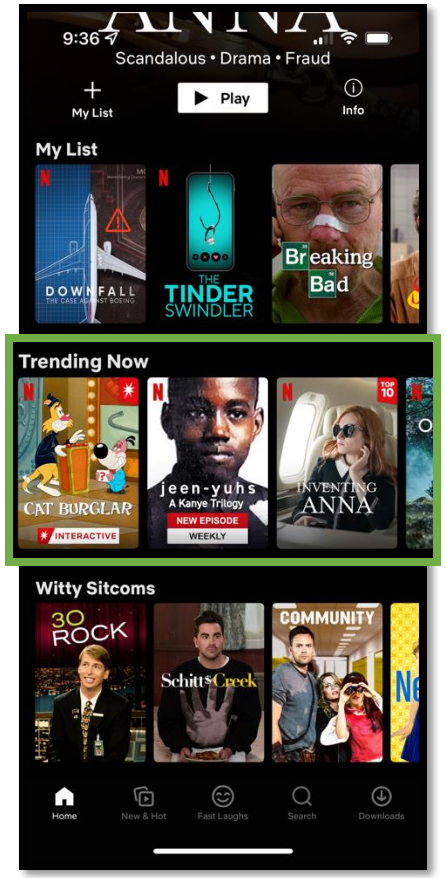
Embracing **partial failure:**

We **do not want to fail** when the bookmarks service is **unreachable** or **producing errors.**



What *should* happen?





Fallbacks:

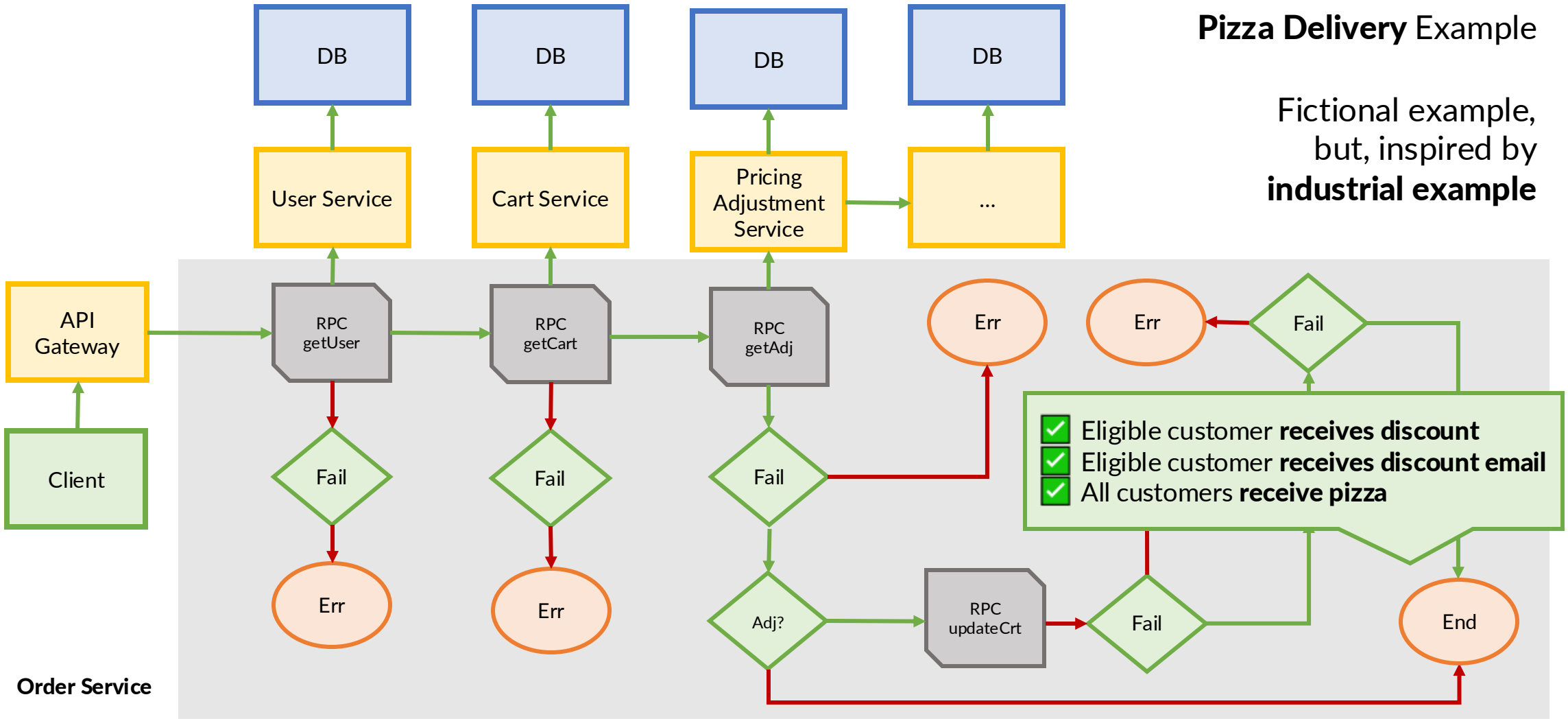
Developers specify **alternative application logic** in the event of dependency failure.

These are called **soft dependencies.**

What ***actually*** happens?

We need to **test** it.

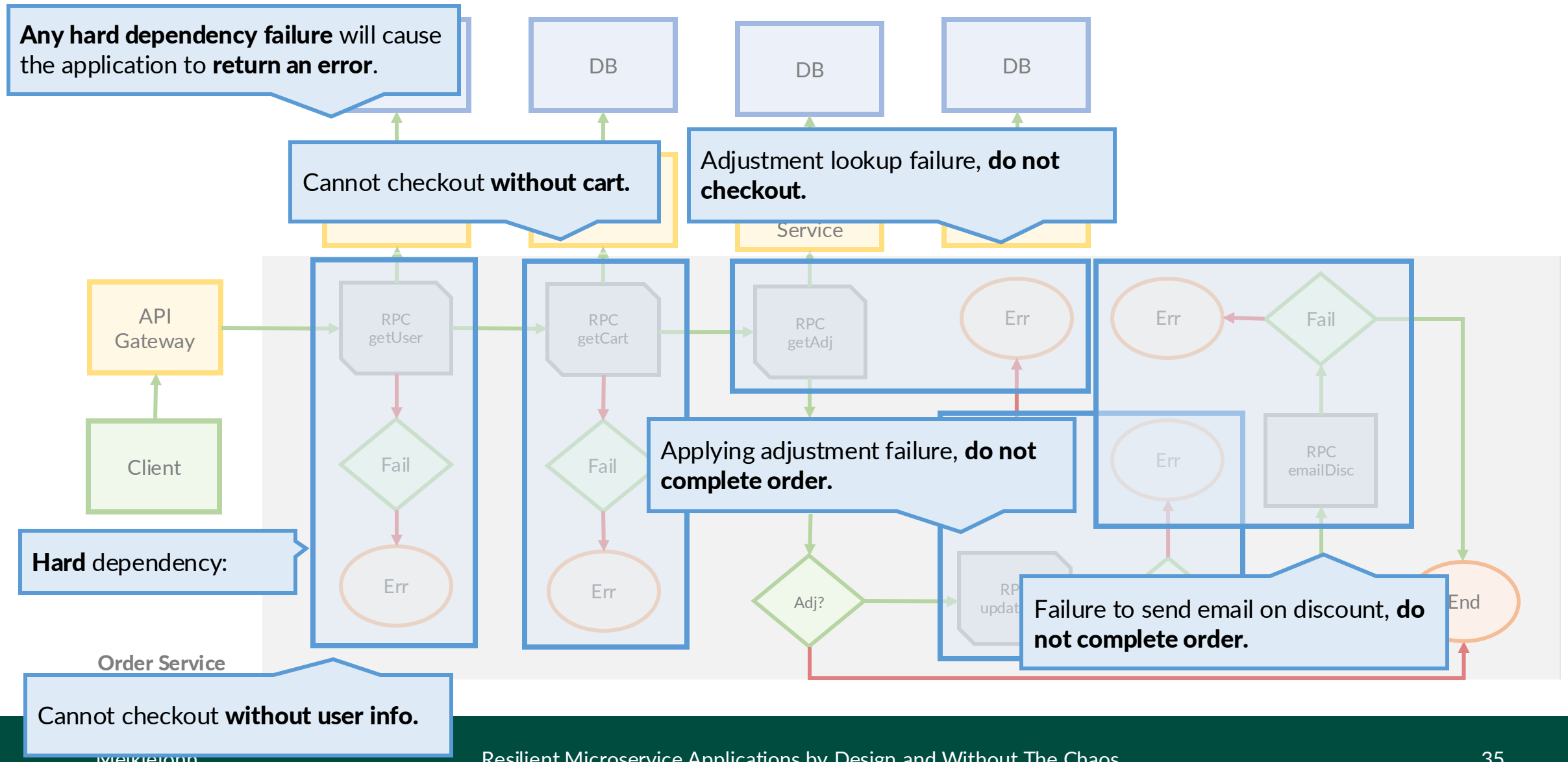
Example: Purchase Application



Pizza Delivery Example

Fictional example, but, inspired by **industrial example**

Purchase: Hard Dependencies



Active Learning: Dependency Types

“Not great.”

“Failure of any dependency forces application to **fail the checkout process.**”

Partner up with you neighbor and **answer the following:**

1.	**What might we want to change about the way this application handles failure?** *(i.e., the business logic, not the application behavior)*
2.	**How will we make sure they are “good” changes?** *(i.e., the business logic doesn’t negatively affect the business.)*
3.	**You guessed it, I’m looking for metrics. What are they?** *(you knew this question was coming.)*

Results of Testing the Application

“Not great.”

“Failure of any dependency forces application to **f**

Business logic decisions conditional on failure **that cannot be automatically determined.**

Identified Problems:

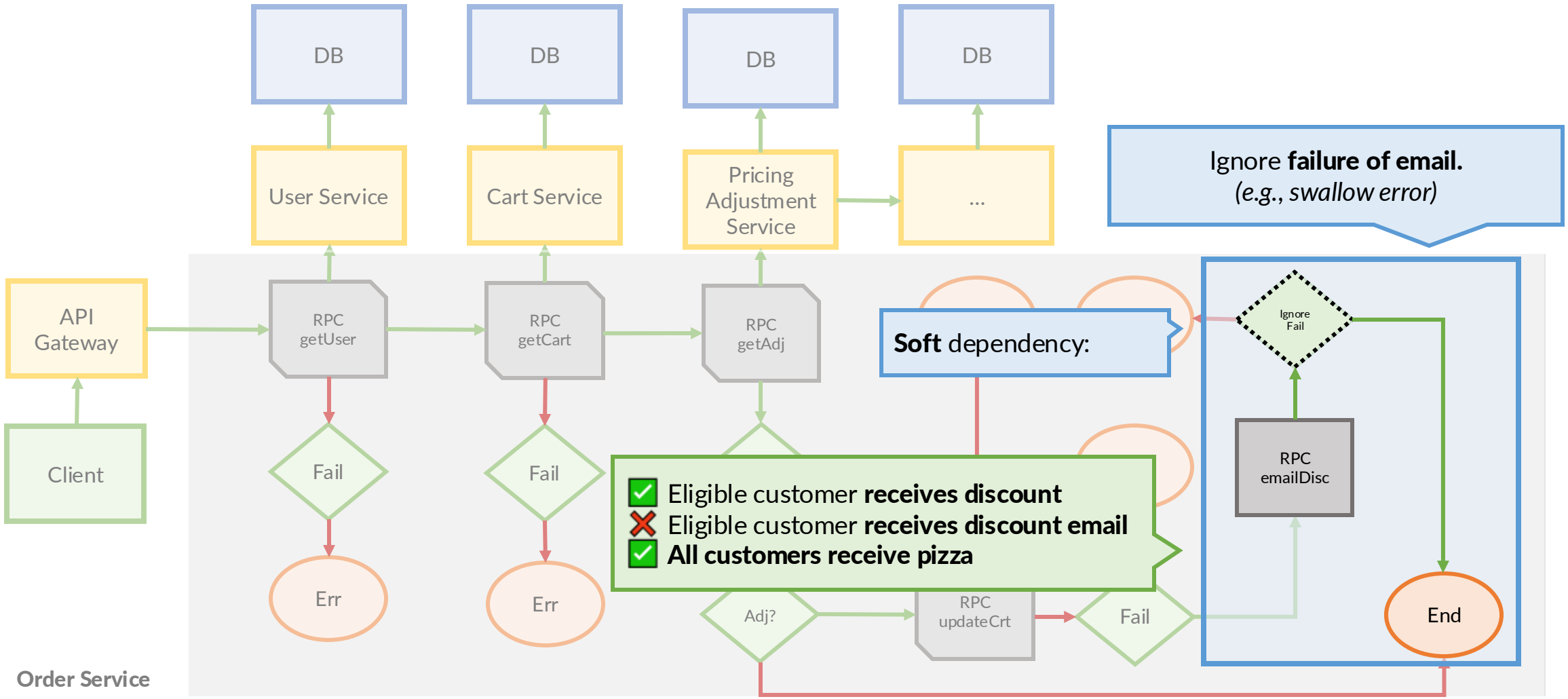
1.	Not being able to **send the discount email** shouldn’t cancel the order with an error.
	*To Fix:* Allow the order to be processed **regardless of email failure.**
2.	Customers **not eligible for a discount cannot checkout if pricing adjustment call fails.** *(where, it would have returned $0, anyway.)*
	*To Fix:* **Assume a pricing adjustment of $0** when the call fails.

Corollary:

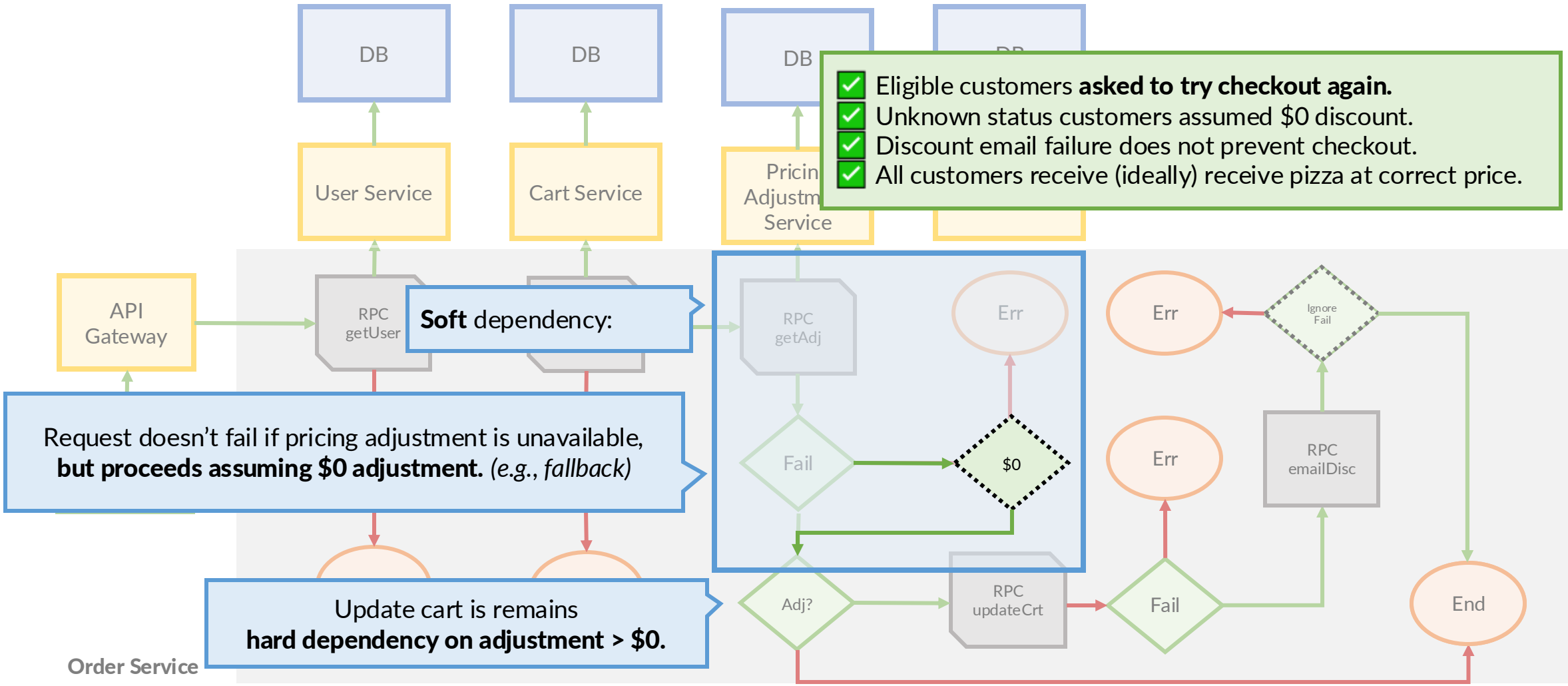
3.	Update Cart (on adjustment > $0) **should continu**
	*Ensure:* ***Ask user who is eligible for an adjustment to try again*** *where the call (may) succeed as user may only be making purchase based on available discount (i.e., first time discount.)*

Cannot reason about the RPC in isolation without understanding the broader context.

Purchase: Ignored Soft Dependency Failures



Purchase: Soft Dependencies with Fallbacks



- ✅ Eligible customers **asked to try checkout again.**
- ✅ Unknown status customers assumed $0 discount.
- ✅ Discount email failure does not prevent checkout.
- ✅ All customers receive (ideally) receive pizza at correct price.

How Can We Do It?

Academic

Formal Methods
Provide specifications of behavior of all services that matched the implementation and verify using a stateless- / explicit-state model checker. *(e.g., TLC, CMC, etc.)*
Significant up-front investment and maintenance due to required specialized knowledge.

Developer-centric, Development-First Resilience Testing
Integrate fault injection testing throughout the development and deployment process.
Fault injection as an extension of the existing functional testing process that also provides guarantees of exhaustiveness with minimal developer overhead.

Chaos Engineering
Randomized fault-injection in production or staging with application observation.
Fails to simulate high-level fault types (e.g., exceptions of a certain type) and gives no guarantee that all failures are covered due to the random nature of the approach.

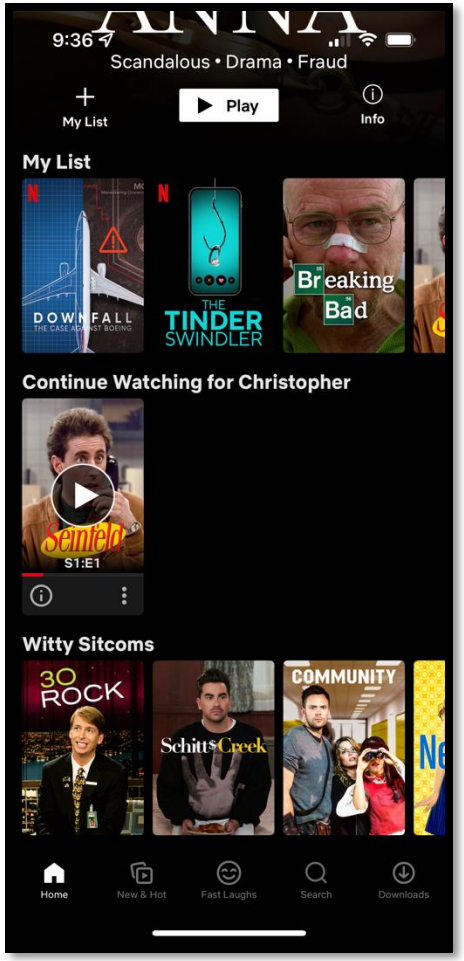
Industry

Where to Start: Simple Mocking

Mocking failure:

Simple mocks for network calls
can simulate failure as well as success.

API Gateway
Service



My List
Service

Bookmarks
Service

Replace with **mock that returns error.**

User Recommendations
Service

Test my API gateway service by **sending it a request to load page.**

Test asserts that **behavior is correct when failure present.**

In Conclusion

Identified the core challenges in making changes to software safely and reliably in a distributed system.

Examined several authorship, testing, and rollout strategies to release code safely.

Practiced identifying problematic changes and how to go about making changes safely.

Any **Questions?**