Reliably Releasing Software

Foundations of Software Engineering

Christopher S. Meiklejohn
Software Engineer
DoorDash

Carnegie Mellon University

DOORDASH

About Me

Started my career in **telecommunications in 1998**, and
I worked in until 2006 (Boston Marathon, Kraft Group) and then
Built Berklee Online (Berklee College of Music) until 2012.

Distributed database development at **Basho and Mesosphere** until 2016
Involved in CRDT research with **SyncFree in Europe**
Production code at at NHS (UK), Rovio (Angry Birds), Riot (League of Legends)
Started my Ph.D. in Europe in 2016 while consulting on distributed systems for **Comcast, Adobe, IOHK, Helium, and Macrometa** before moving to Carnegie Mellon University in 2018.

Intern'd at **Microsoft Research (3x) and Amazon's Automated Reasoning Group** working on serverless (Durable Functions) and formal methods (S3.)
Finished Ph.D. in Software Engineering focusing on building reliable microservice applications in May 2024 at **Carnegie Mellon.**
(TA'd and co-instructed 15-313.)

Software Engineer at DoorDash working on Order Platform focusing on reliability and the future of Order Platform.

Goals

Identify the core challenges with modifying, testing, and deploying applications safely in a microservice architecture.

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

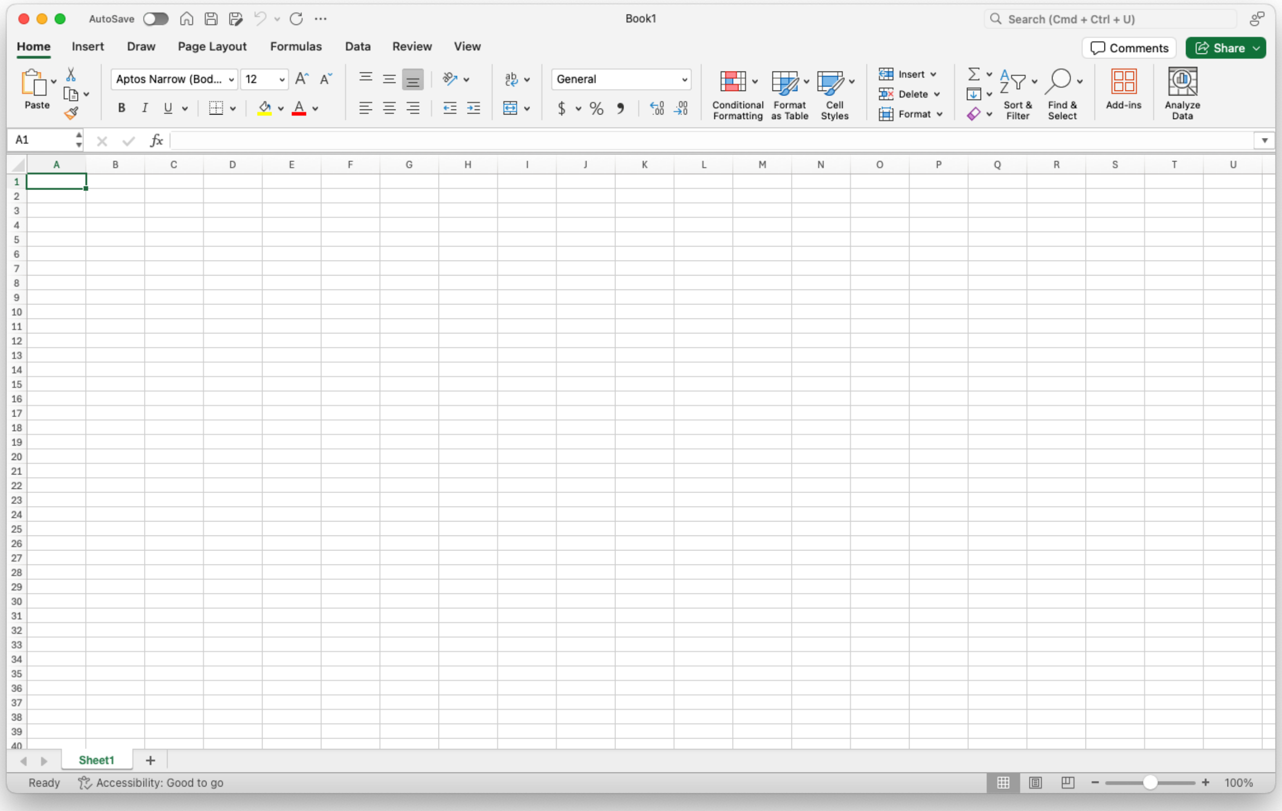
Practice authoring a safe rollout plan for a new feature in a microservice application.

How Do You Change This Software?

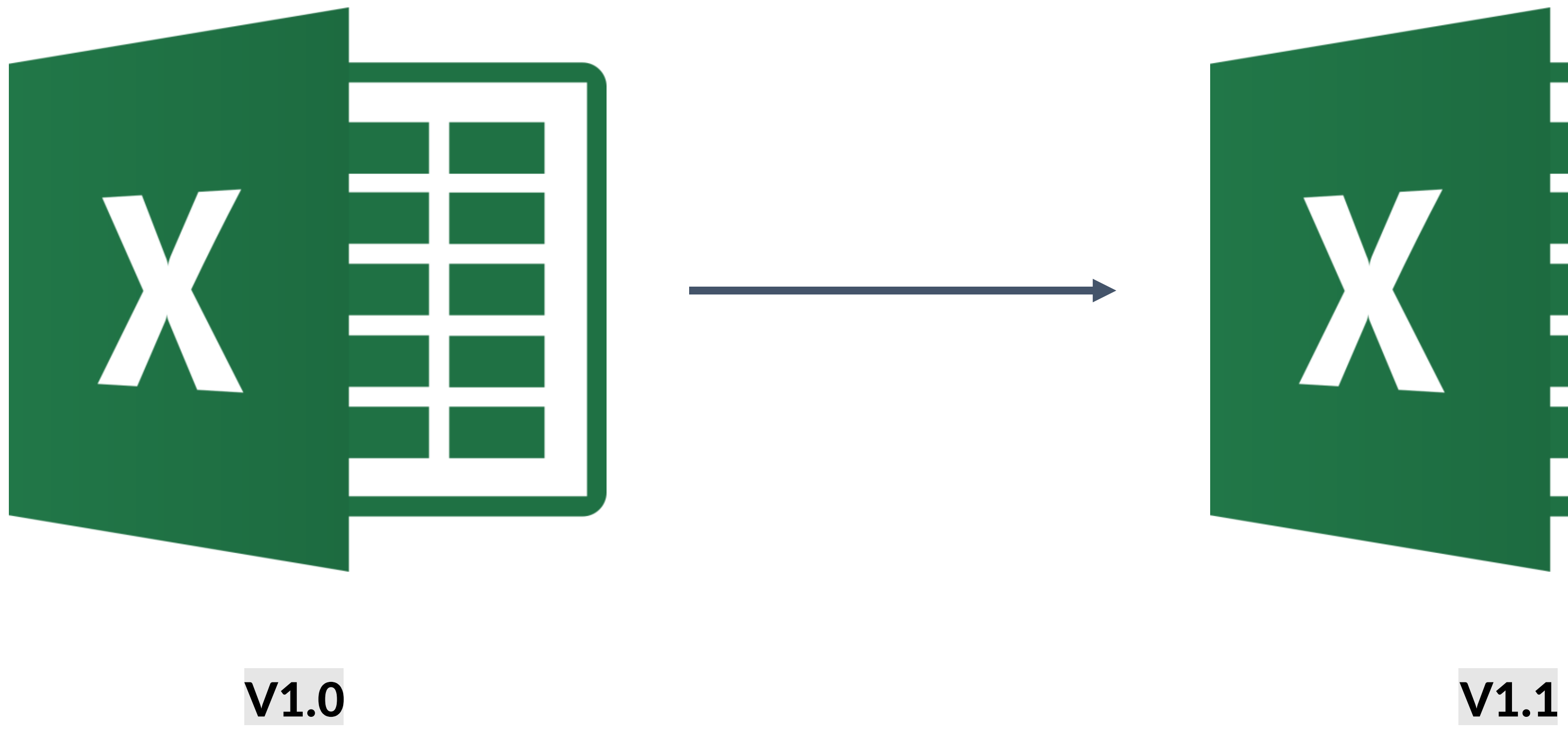
Modify
Implement one or more changes in the application and build the new version of the application.

Test
Test the application using a test suite or QA process to ensure application works correctly.

Release
Create new version of the software, users close their existing version and install it and open the new version.



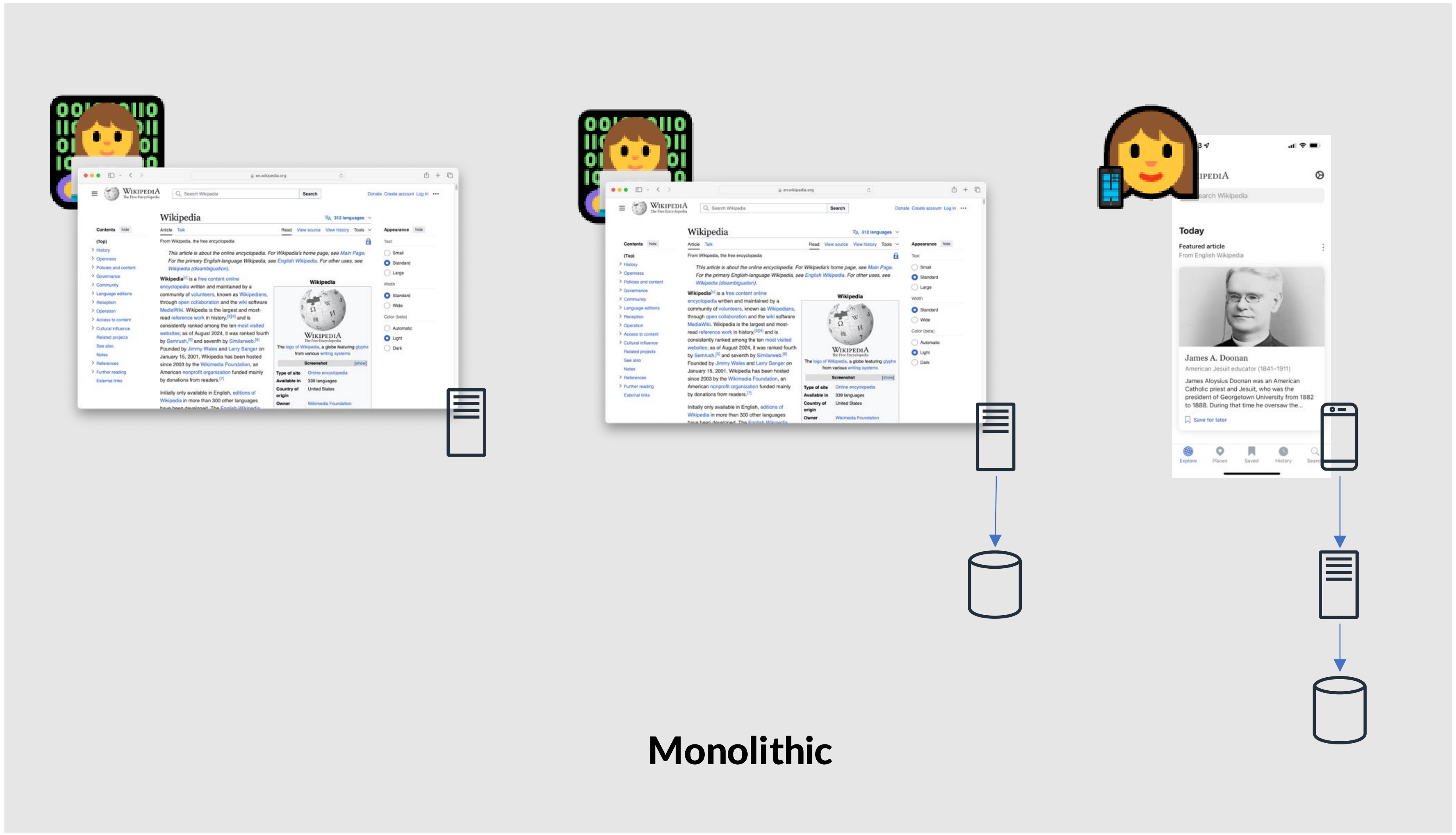
App Upgrade: One Version To The Next

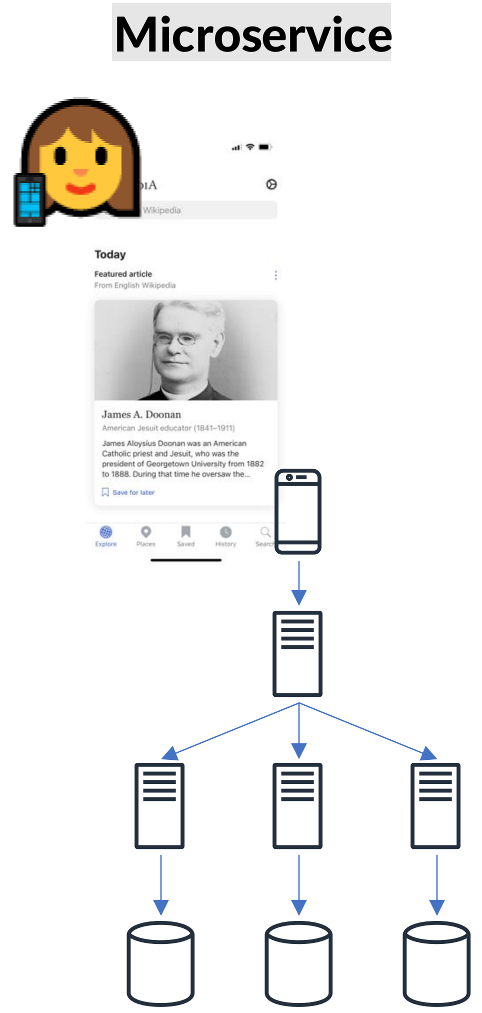


Similarly, if we want to **scale up this application to more users**, we just have users **install more copies of this application on *their* computer.**

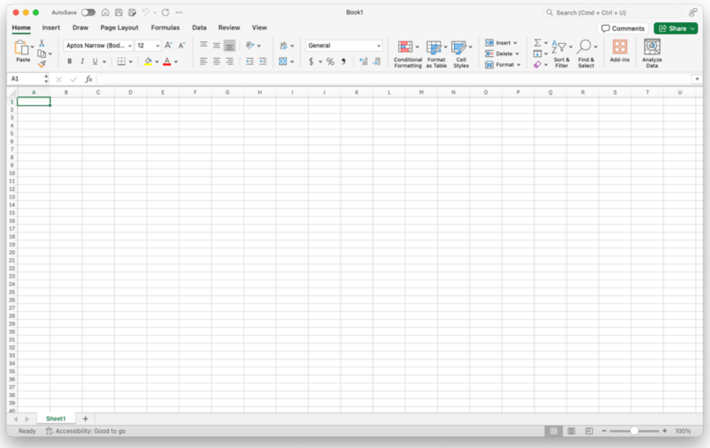
This detail will **become important later.**

What About This Software?





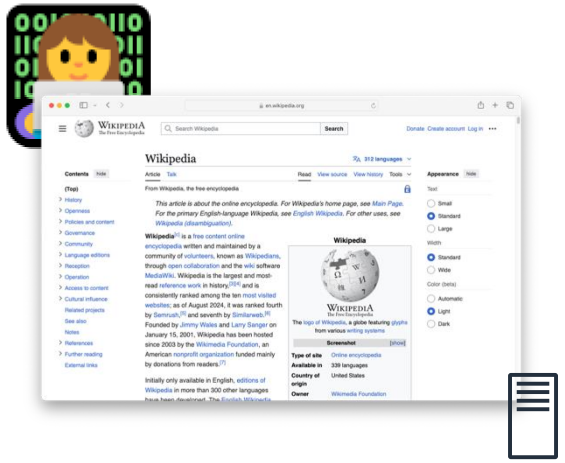
What Are The Differences?



Location
Application runs on a server, and is not installed on client’s device.

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

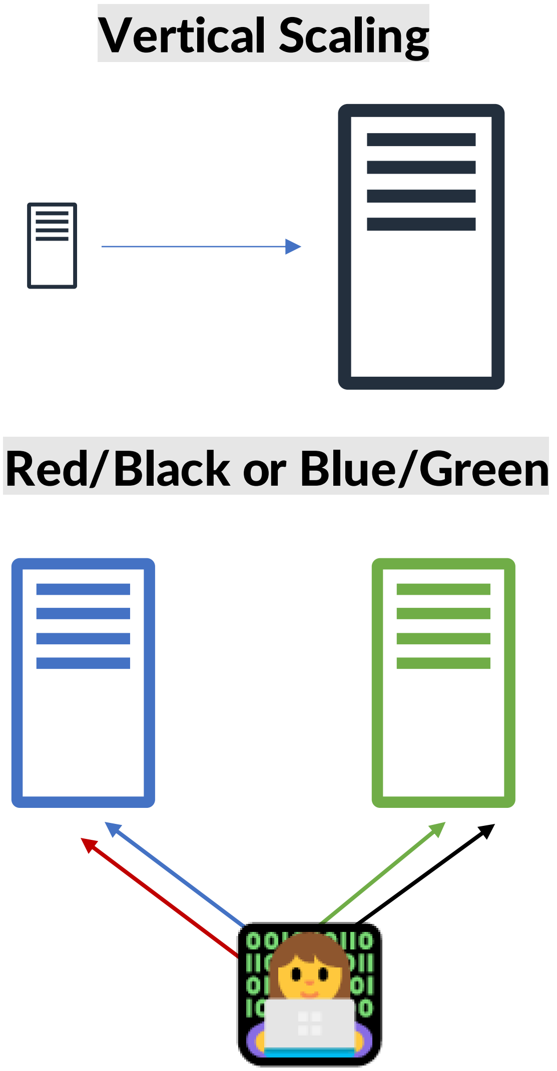
“Always On”
Applications are upgrade in place, typically aiming for zero-downtime.

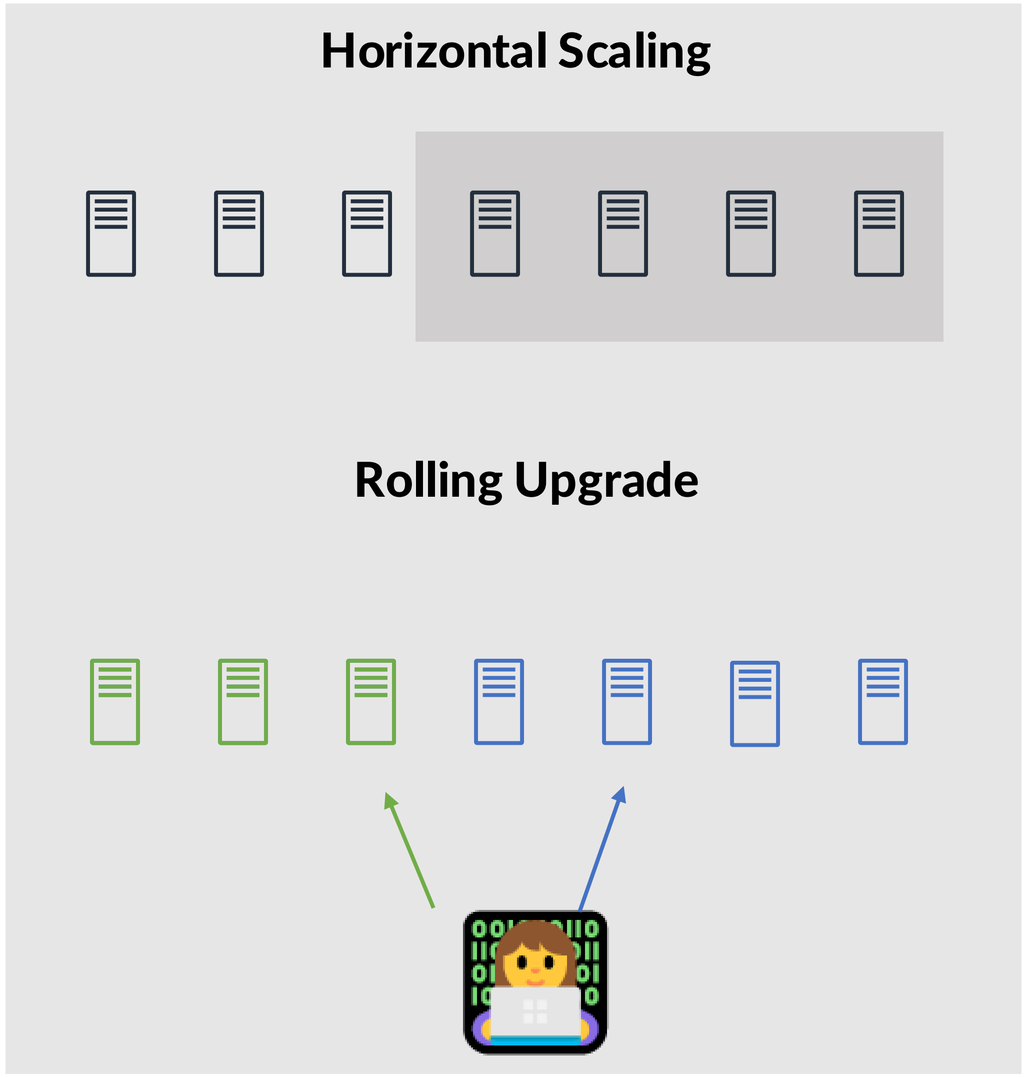


Scaling and Deployments: Intertwined

Scaling

Deployment





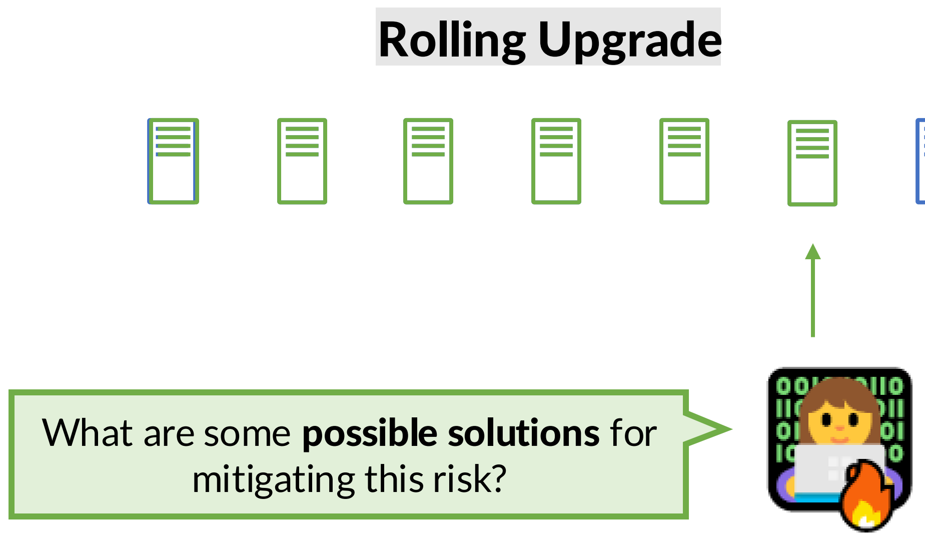
Bugs?

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.


Rolling Upgrade
What are some **possible solutions** for mitigating this risk?

Dark Launch

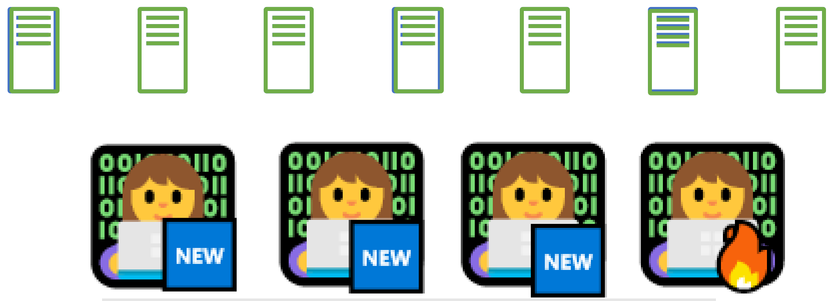
Solution: **Dark Launch**

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!**

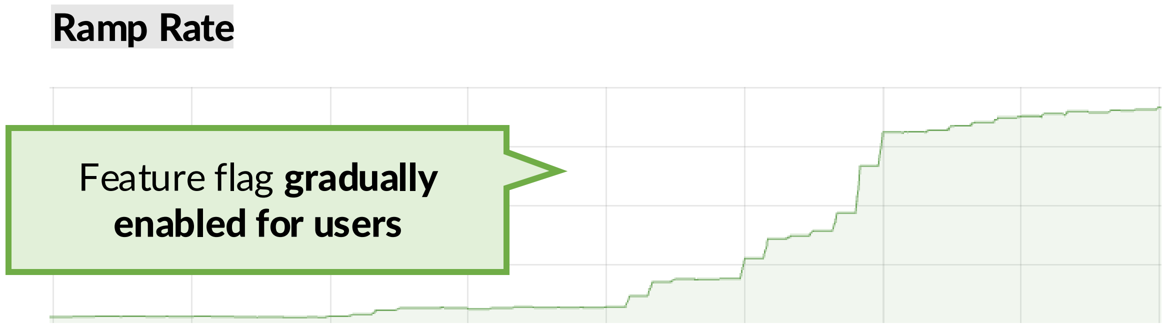
Dark Launch: Observability

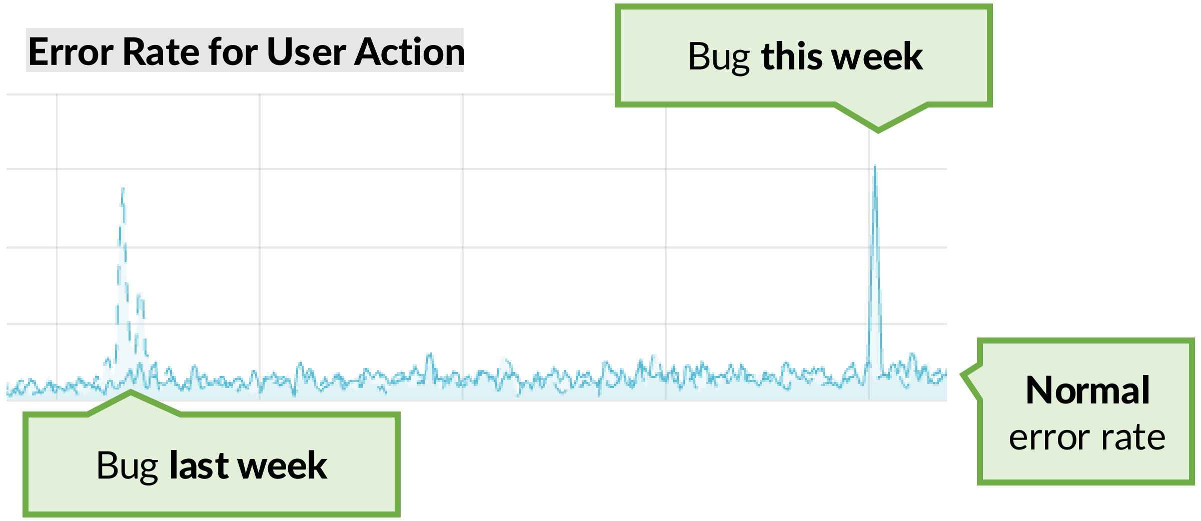
How do you **identify a rollout problem?**

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.





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.**

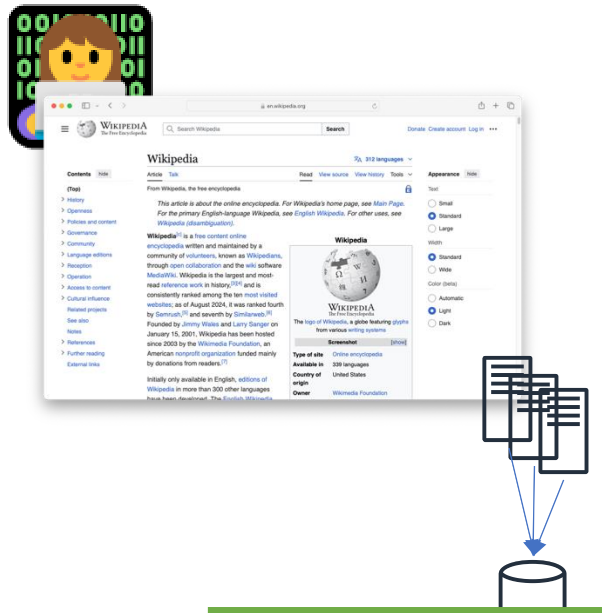
   Rate of users who see priority option, select it, but to not receive a priority ride when available.

   Rate of users who should see priority option, but do not see the priority option.

What About This?

Modifications to DB + App
Many times you will have to modify the database with the application for new features.

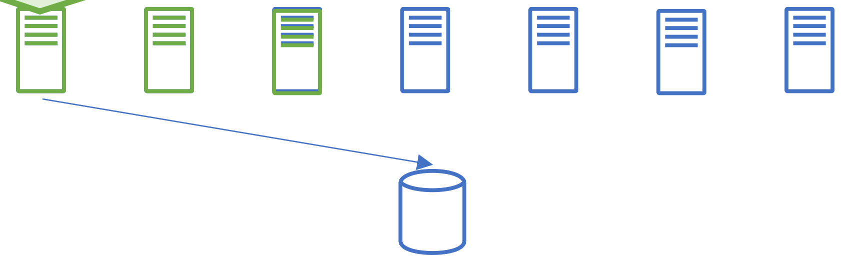
No Rolling Upgrades
Can't synchronize rolling upgrade between app + database, no rolling upgrade for DB, even schema changes in distributed databases are atomic across nodes.

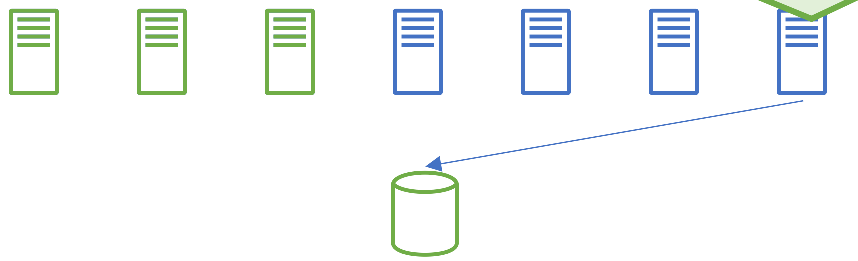


New version might be **incompatible with old DB.**

Problems During Rolling Upgrade/Release

Old version might be **incompatible with new DB version.**





Active Learning: Database Changes

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

1. **What types of database changes can be safe in isolation?**
   (i.e., without requiring modification of the application)

2. **Are there any application changes, which require DB changes, that are safe?**
   (i.e., application will rely on DB change, but rollout coordination is not needed.)

3. **How can I safely add a field to the database, that the new version of my application will use?**
   (hint: use the techniques we've already discussed to figure out how to do this safely.)

Database Changes: Adding a New Field

1. **Add new field to the database using a migration.**
New field added to the schema, but nothing uses it.
Nothing *(i.e., indexes, integrity constraints, etc.)* can use this field and field **must be nullable.**

2. **Dark Launch Application With Code To Write Field**
Dark launch new version of application with code to begin writing the new field.
Gradually roll out feature that writes the new field.

Code to write field **may contain a bug** *(e.g., serialization.)*

3. **Dark Launch Application With Code To Read Field**
Dark launch new version of application with code to begin reading the new field.
Gradually roll out feature that writes the new field. **Must handle nulls!**

Code to read field **may contain a bug** *(e.g., logic error.)*

Only after you've rolled out features to 100% of all users and waited for bug reports:

4. **Remove Migration Code**
Deploy version of code without migration *(i.e., feature flags.)*
You can't dark launch this, otherwise you'll loop indefinitely.

Exercise: Database Queries

Bob added a new field to the database **following all of the best practices.**

1.	Added the column and dark launched code to begin reading the new column.
2.	Ran a database migration off hours to back populate the new column.
3.	Ramped up code to begin reading the new column.
4.	Dark launched and rolled out code to stop reading old column by searching in the code for SELECT/UPDATE statements that read the column.

Now, Bob is asked to remove that column to save storage costs:

Bob removes column and site **immediately fails to process any user requests because a query is still using that field. Bob can't rollback because a DROP COLUMN is destructive.**

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

1.	**How could Bob have reduced the risk of removing the database column?**
2.	**Bonus Question: What went wrong?**

Database Queries

What was **the bug?**

It was a **SELECT * statement** that **did not directly reference the column name**; however, the code did reference the name much later, and in a different location making it difficult to search for the identifier in a large code base.

How could this **destructive, not-backwards-compatible** change be done **safer?**

Using **tiered deprecation:**

1. **Rename the column, in order to find undetected usages but preserves rollback possibility.**
2. **Drop column after renaming, which reduces risk, but does not eliminate probability, of usage.**

This strategy can be **applied everywhere**: a number of incremental changes to identify usage of APIs before making final destructive change.

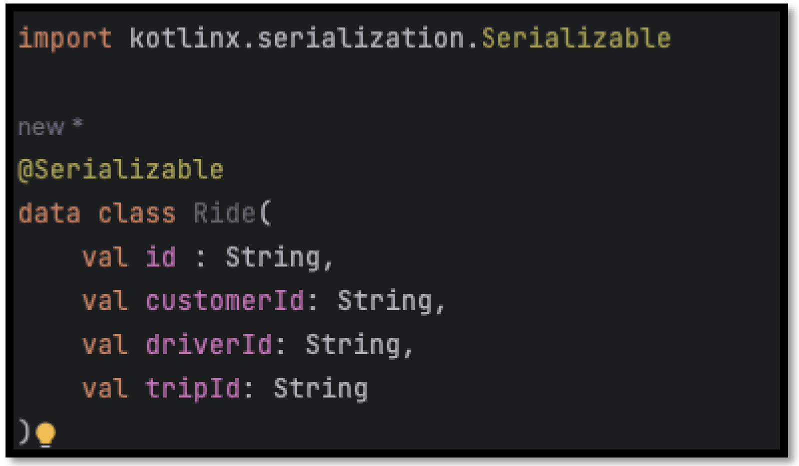
Exercise: Data Serialization

Ride is an object that's **serializable** and written into the database.

Alice is making a change to add a **priority** column, a Boolean, to indicate whether the ride is a priority ride or not.

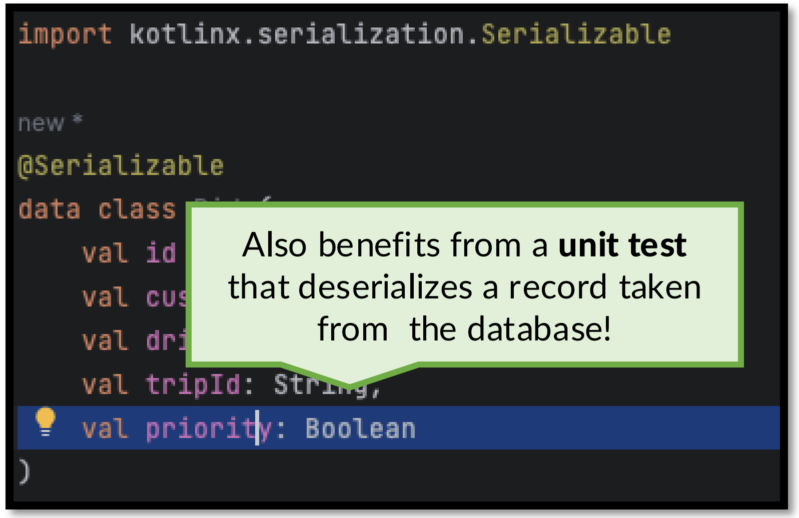
Partner up with you neighbor and **answer the following:**

1.	**What is wrong with this change?**
	Deserialization will fail on all Ride's that do not contain the priority field.
2.	**How could this be done safer?**
	Value must be provided with a default value or allowed to be null.



```kotlin
import kotlinx.serialization.Serializable

new *
@Serializable
data class Ride(
    val id : String,
    val customerId: String,
    val driverId: String,
    val tripId: String
)
```



```kotlin
import kotlinx.serialization.Serializable

new *
@Serializable
data class
    val id
    val cus
    val dri
    val tripId: Str   ng,
    val priority: Boolean
)
```

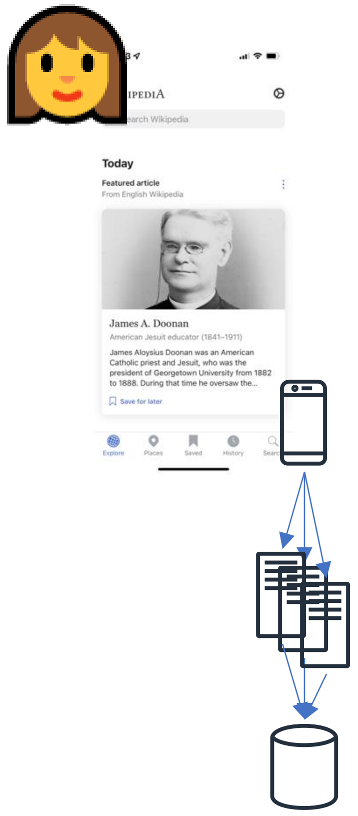
Also benefits from a **unit test** that deserializes a record taken from the database!

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.**


IPEDIA
earch Wikipedia
Today
Featured article
From English Wikipedia
James A. Doonan
American Jesuit educator (1841–1911)
James Aloysius Doonan was an American Catholic priest and Jesuit, who was the president of Georgetown University from 1882 to 1888. During that time he oversaw the...
Save for later
Explore
Places
Saved
History

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

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: Testing and Deprecation

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.

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.

Microservice Architectures

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
As of 2021, Netflix had 1,000 services in their microservice platform used to deliver streaming services.

Uber
As of 2016, Uber had 2,200 services in their microservice platform. 120 services were involved in obtaining a ride share as a consumer.

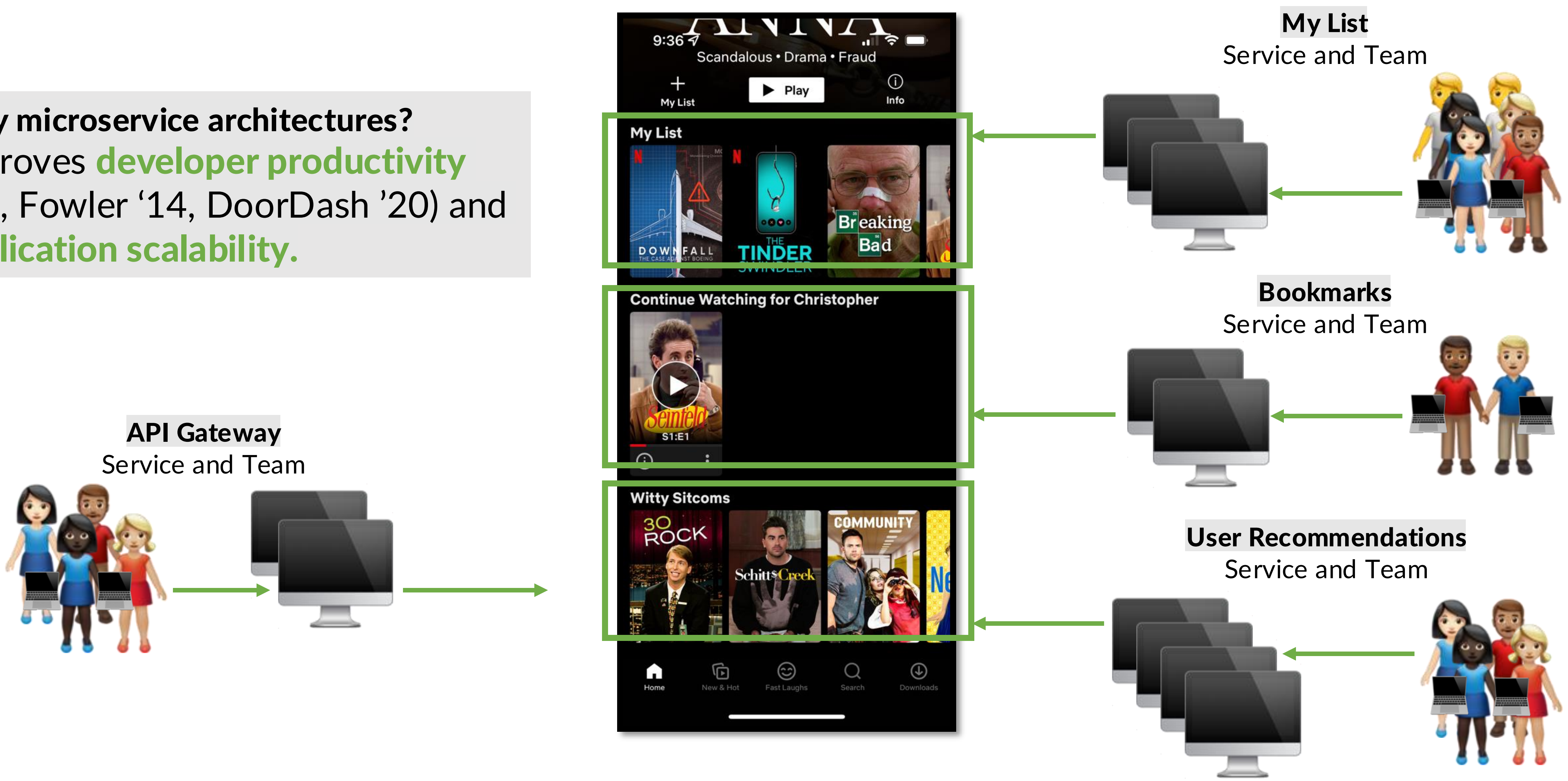
DoorDash
As of 2022, DoorDash has over 100 services in their microservice platform used to provide food delivery and ordering services.

As of 2021, **all 50 companies in the Fortune 50** were hiring for roles that **mentioned microservices.**

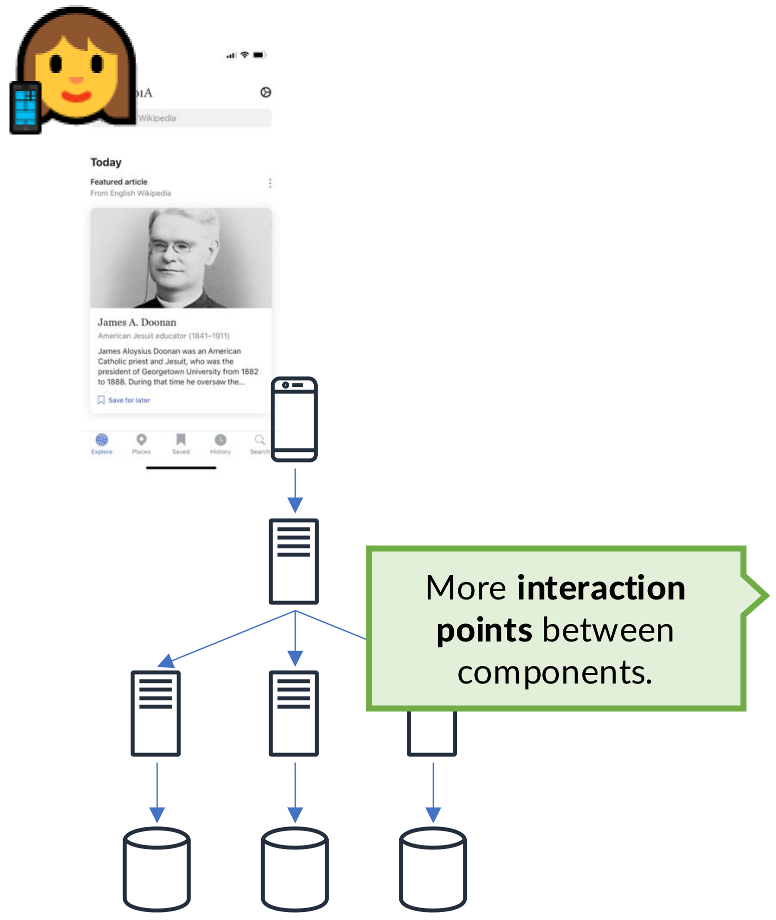
Microservice applications are the most common and complex type of distributed application being built today.

Netflix: Microservice Architecture

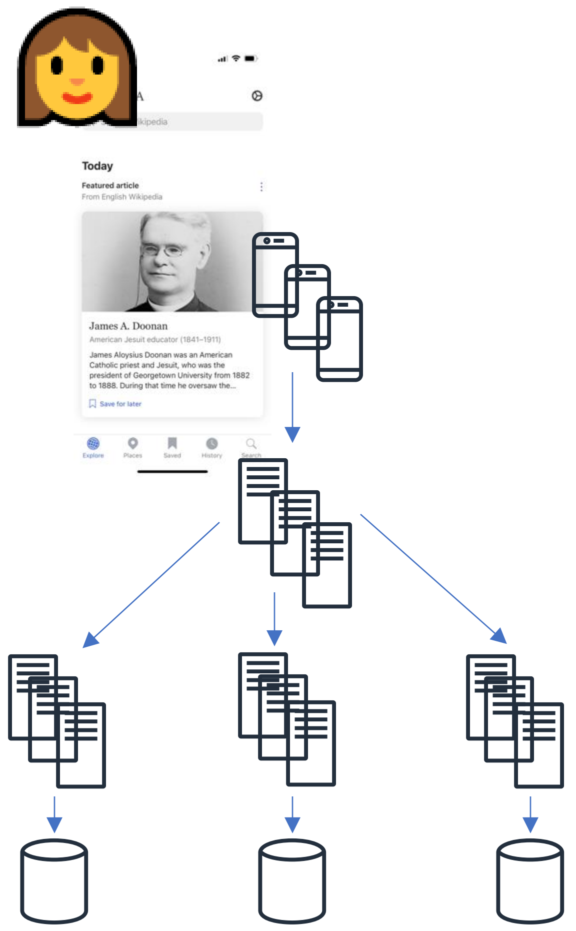
Why microservice architectures?
Improves **developer productivity** (*e.g.*, Fowler '14, DoorDash '20) and **application scalability.**

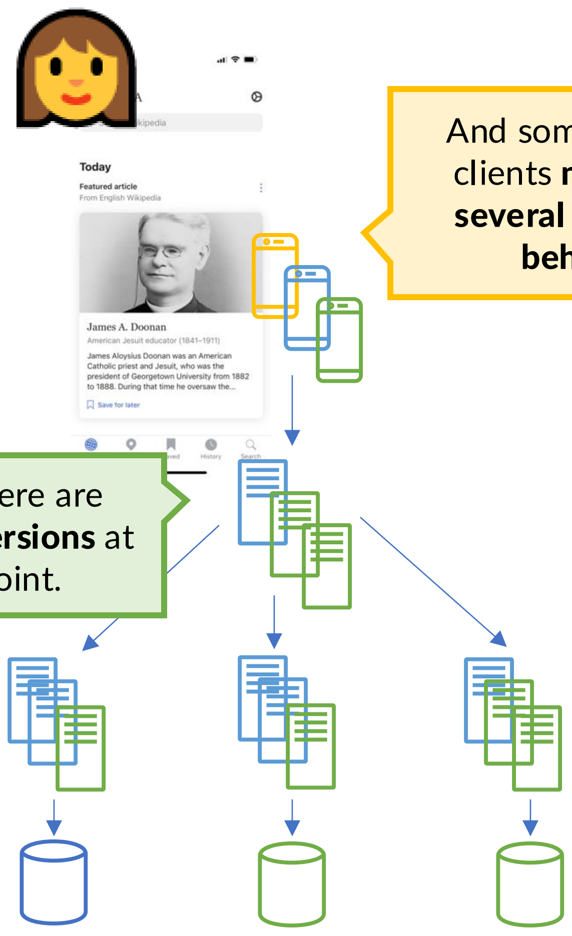


What About This?



More **interaction points** between components.





Where there are **different versions** at each point.

And some mobile clients **might lag several versions behind.**

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.)**

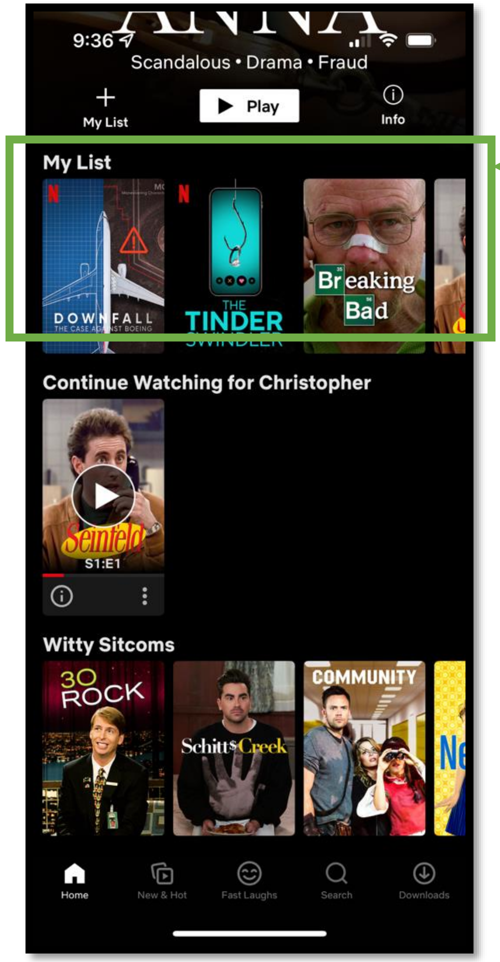
Anyone who calls you *(i.e., "upstreams")* **must run tests for all features that touch that with all combinations of feature flags (theirs, and yours) whenever you make a change.**

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

Netflix: Testing

…and, I might even need to **test the API Gateway with the Mobile App to make sure it's still compatible** if my change to My List altered the API Gateway's response format.

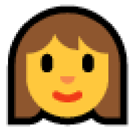
API Gateway
Service and Team



My List
Service and Team

If My List makes a change, **I have to make sure it's still compatible with the API Gateway through testing.**

This is intractable,
so how can we make progress and get work done?



Testing: Knowing Your Service

First 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 in a call graph)*
   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.

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.

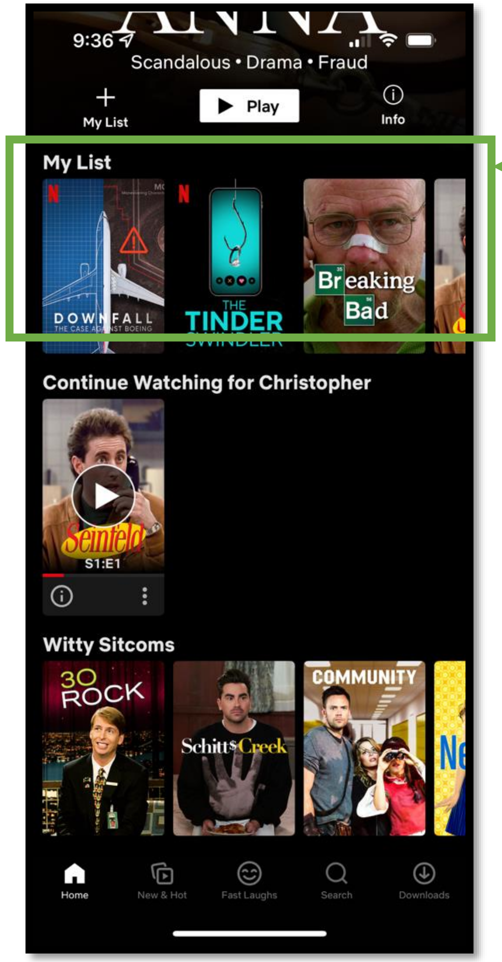
If all tests passed, no tests were added, **but mock drift detected:**

1. Missing **necessary test coverage** for a new feature.
(e.g., field has no assertions)

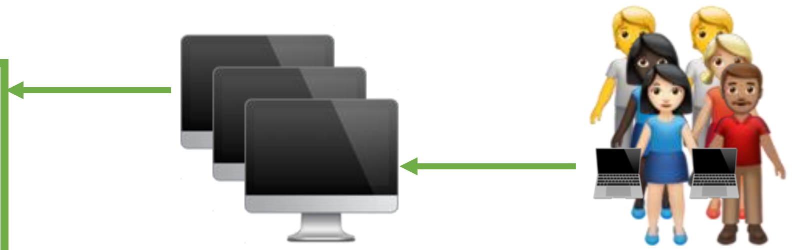
2. Data is being **passed through you to an upstream caller and not used directly by your service.**
(e.g., field has no assertions)

"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



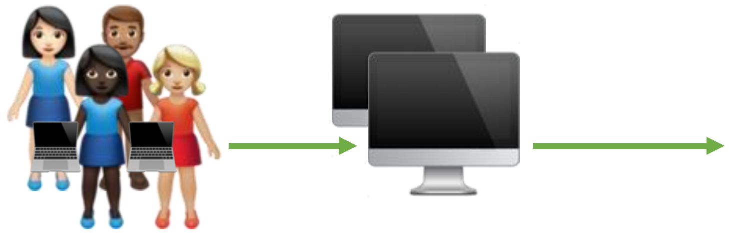
My List
Service and Team



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.

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.

In Conclusion

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

Examined a number of authorship, testing, and rollout strategies to release code safely.

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

Any **Questions?**