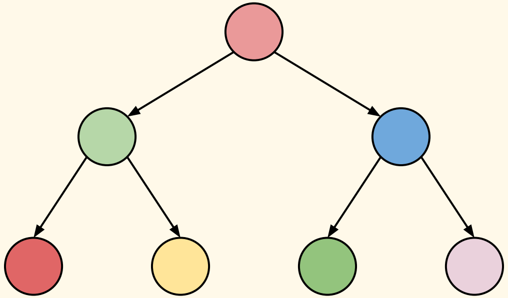
Keeping Master Green at Scale

Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel,
Balaji Varadarajan, Simon Soriano, Dhaval Patel, Ali-Reza Adl-Tabatabai

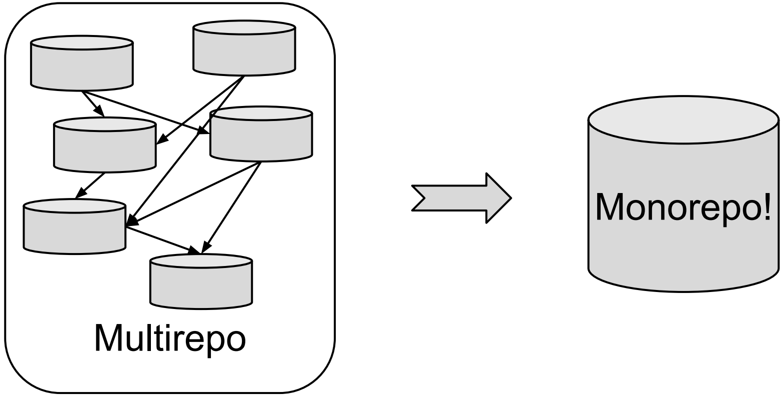
(https://eng.uber.com/research/keeping-master-green-at-scale/)



Uber

Monorepo is popular!

- Single, shared repo hosting companies' software assets



Advantages of a Monorepo [Ciera et al. @ICSE'18]

- ✔ Simplified Dependency Management
- ✔ Improved Code Visibility

Always green master considered hard

- **Monorepos handle a huge volume of commits every day**

- **Existing CI workflows do not guarantee an always green master**
  - Too hard at scale

- **Submit Queue guarantees an always-green master at scale**

UBER

Outline

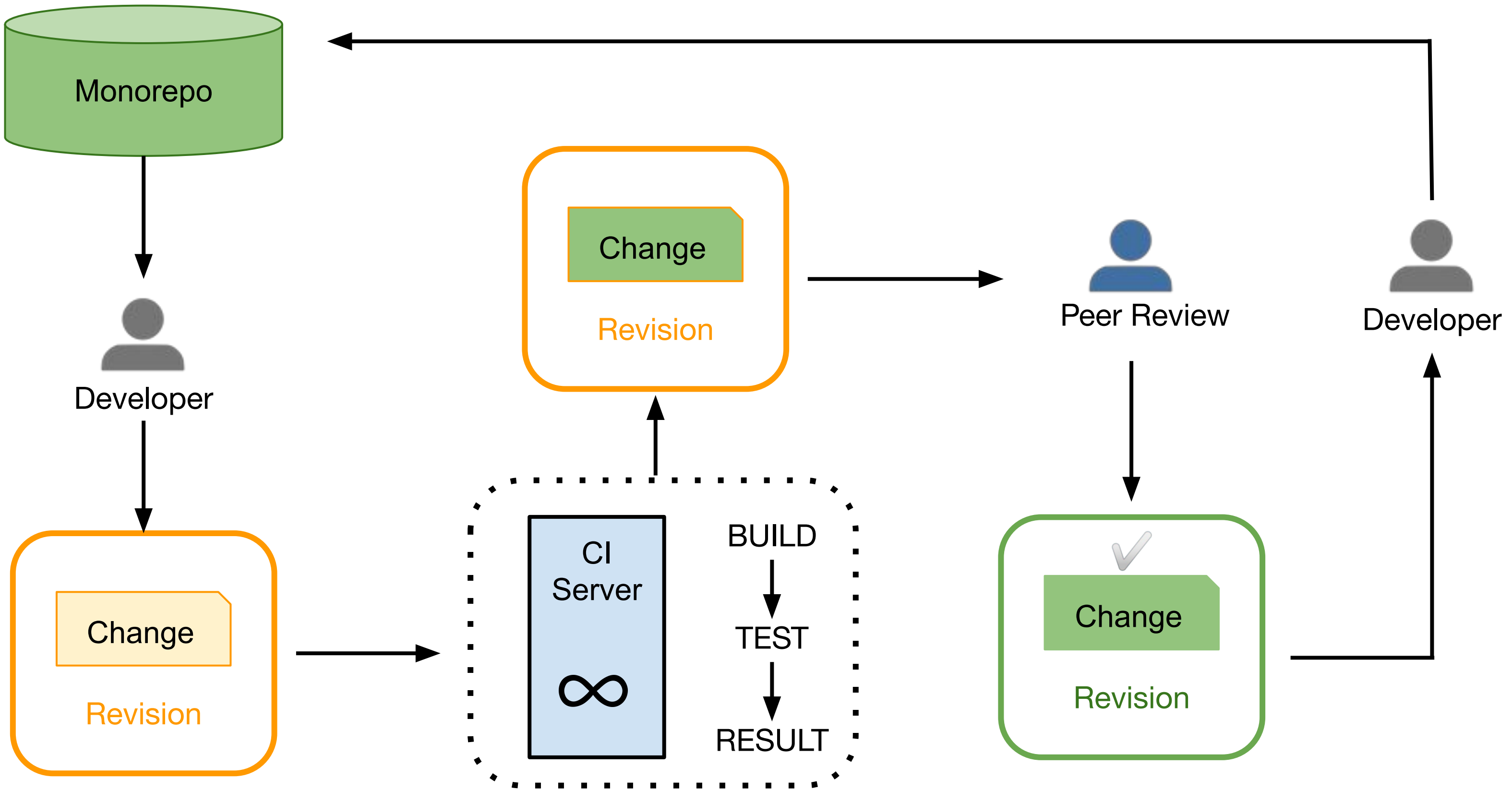
01 Why green master is hard

02 Probabilistic Speculation

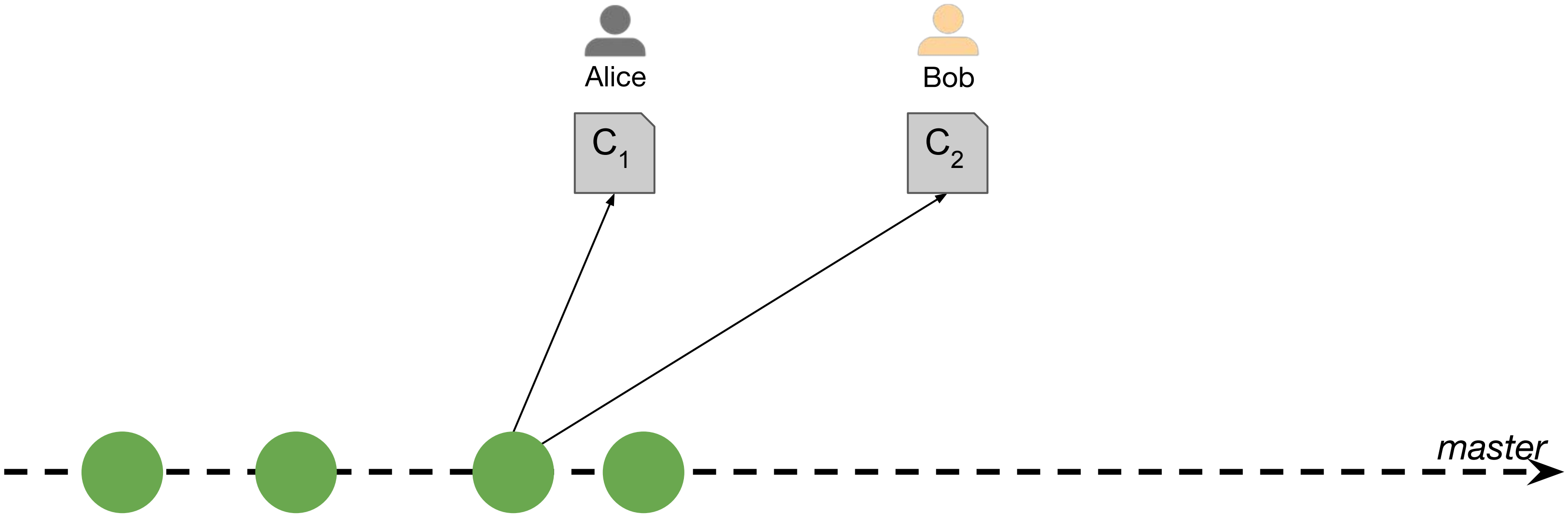
03 Conflict Analyzer

04 Evaluation

Lifecycle of a change in monorepo

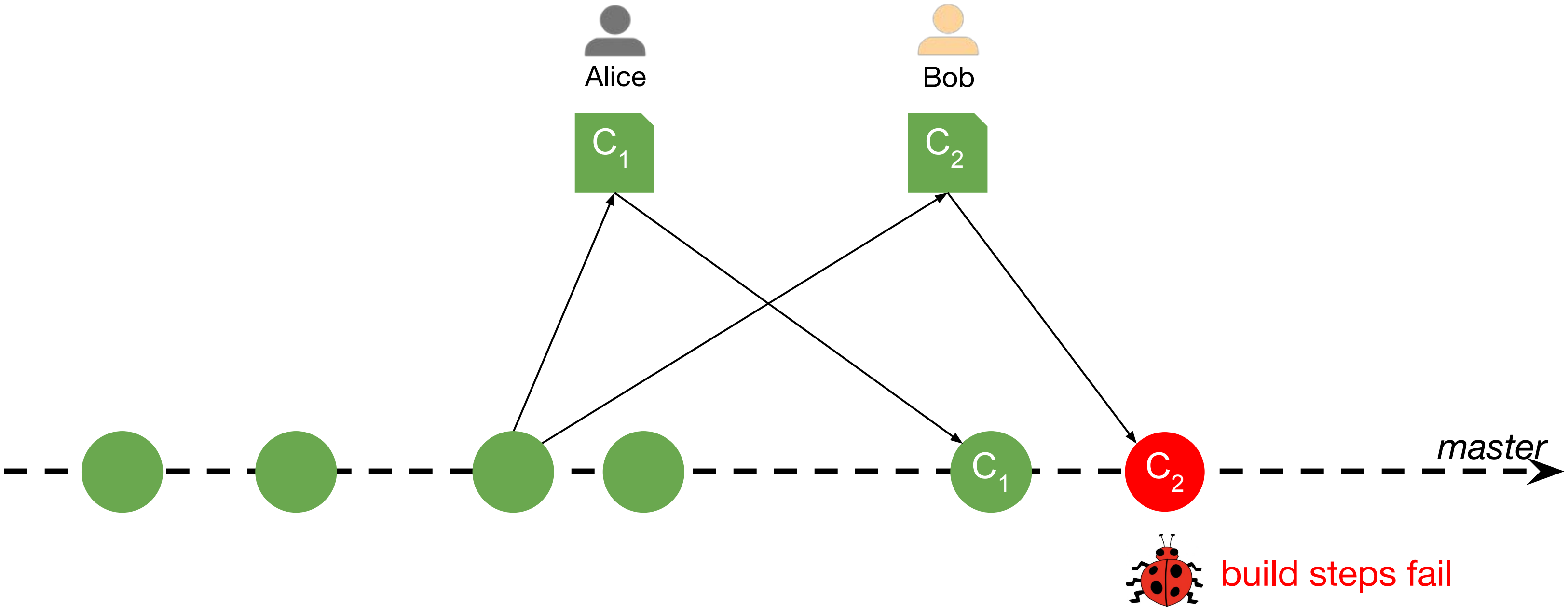


Challenge: Concurrent conflicting changes



UBER

Challenge: Concurrent conflicting changes



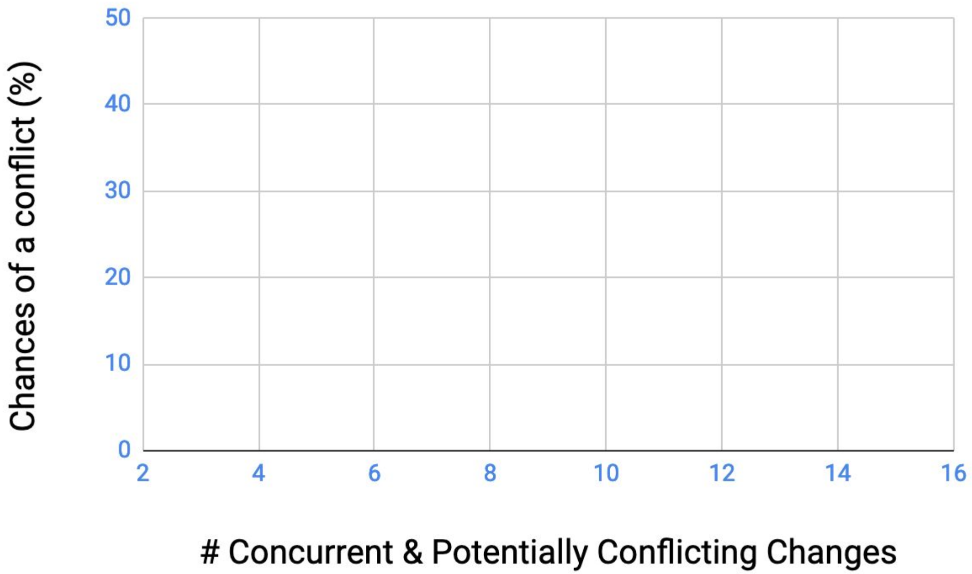
UBER

Example of a real conflict

```
@@ -287,6 +288,9 @@ + (void)registerProtocols
     [currentGraph registerProtocol:@protocol(AnalyticsDeveloper)
                 withImplementation:analyticsManager];

+    DependencyProvider *theProvider = [DependencyProvider theProvider];
+    [theProvider setAnalytics:analyticsManager];
+
```

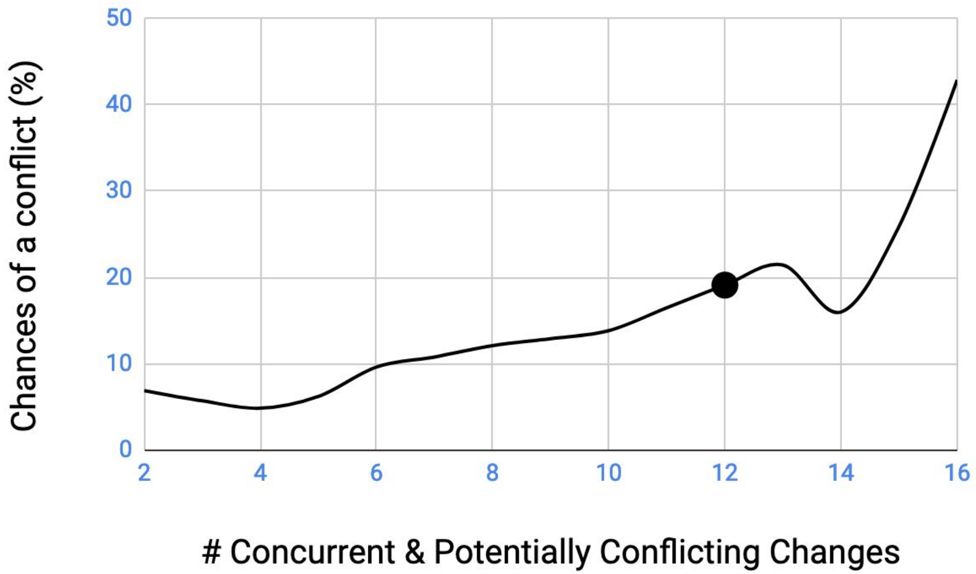
```
@@ -377,10 +378,14 @@ + (void)registerProtocols

 #pragma mark - Networking
     [currentGraph registerProtocol:@protocol(NetworkingConfiguration)
-                       withFactory:^id {
-                           return [[NetworkingConfiguration alloc] init];
-                       }];
+                withImplementation:networkingConfiguration];
+
+    HTTPProvider *theProvider = [HTTPProvider theProvider];
+    [theProvider setLogger:logger];
```

How often conflicts happen?



UBER

How often conflicts happen?



Observation: Chances of a conflict ↑ from 5% to 40% as #. of concurrent & potentially conflicting changes ↑

UBER

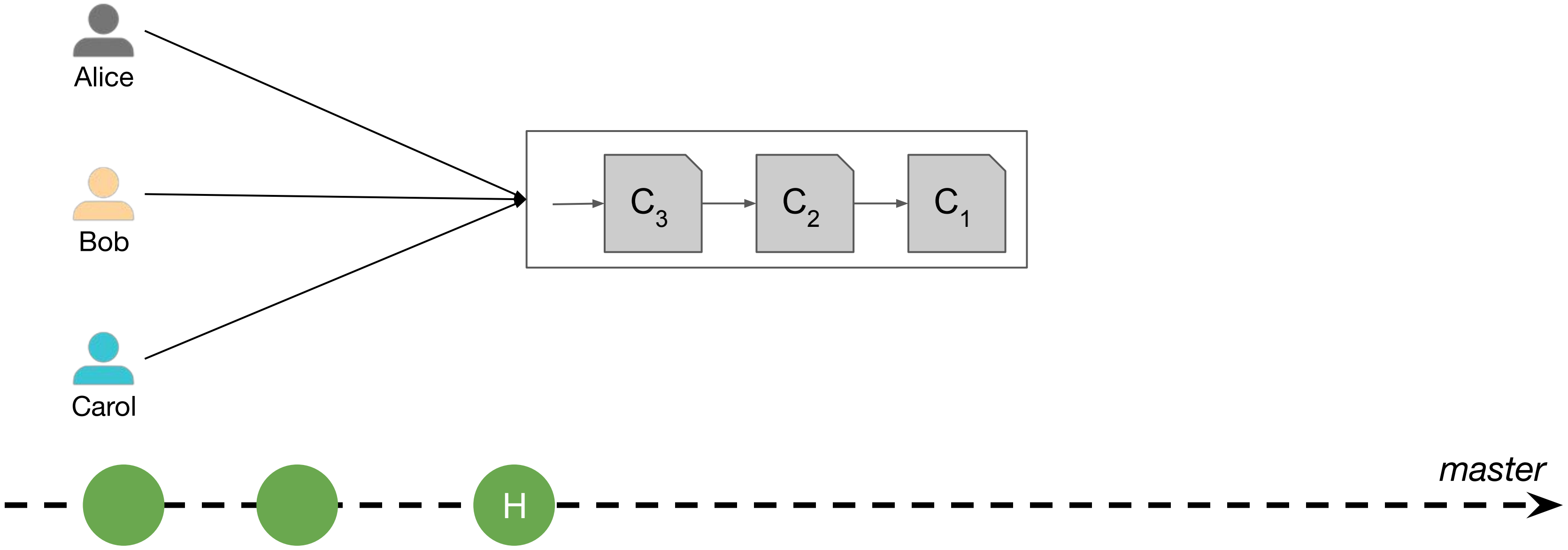
Drawbacks of a red master

Delayed rollouts

Hampered Productivity

Complex rollbacks

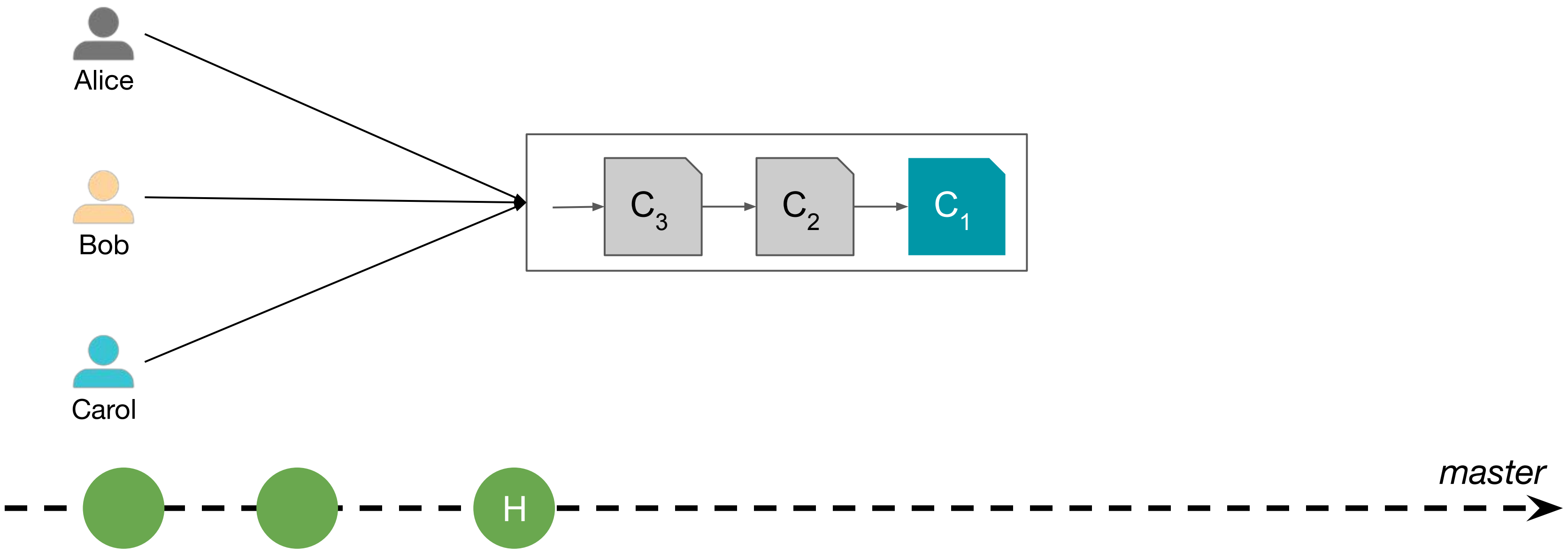
Keeping master green: Queue



Alice, Bob, Carol enqueue changes they want to commit

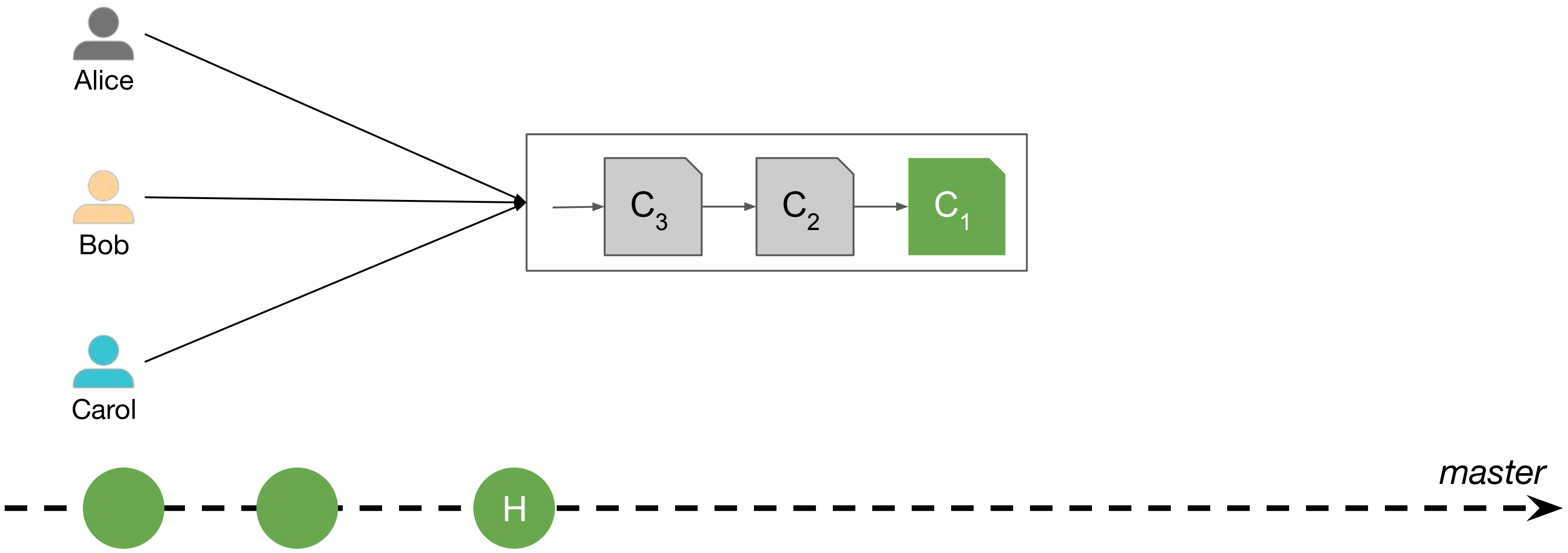
UBER

Keeping master green: Queue



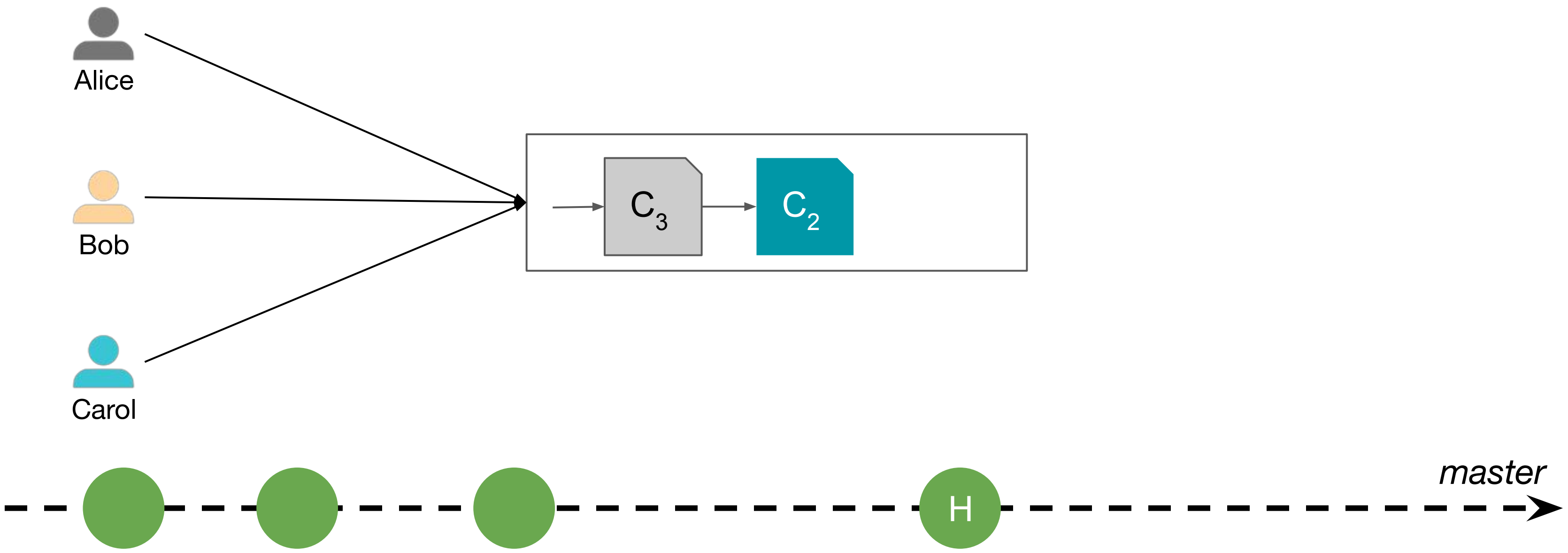
C_1 is built and tested against mainline head (H).

Keeping master green: Queue



Build steps for $H \oplus C_1$ succeed.

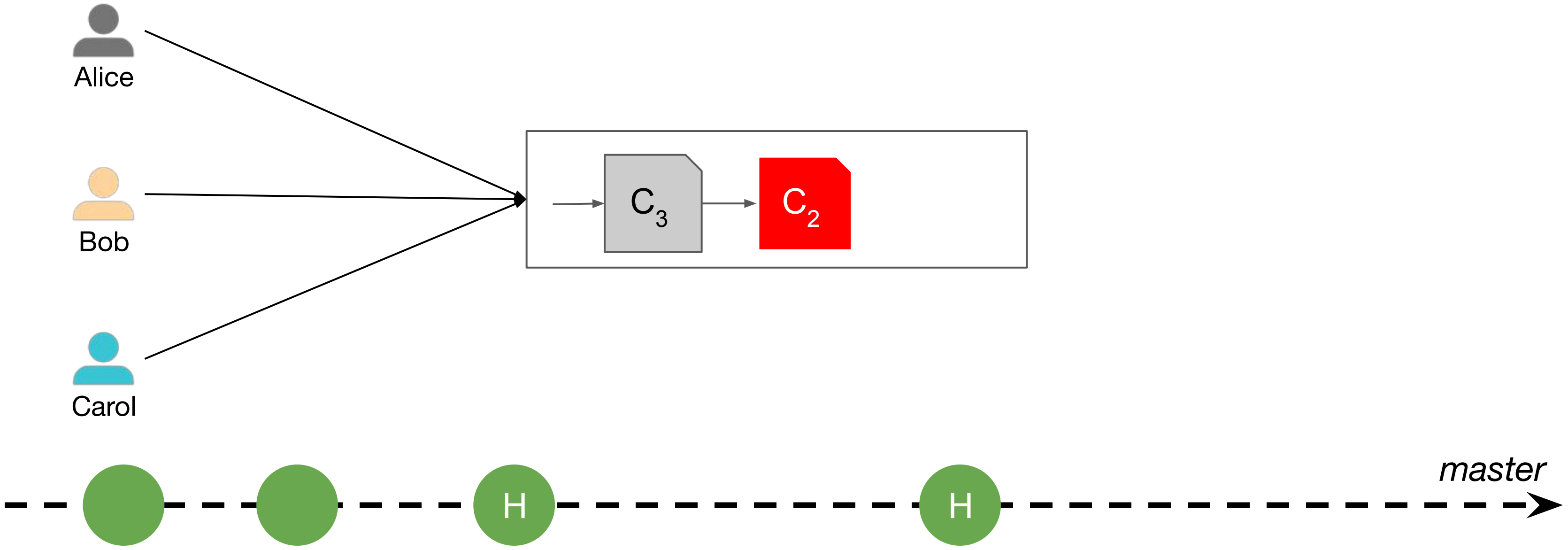
Keeping master green: Queue



C_1 is committed and it becomes the head. C_2 is tested against it.

UBER

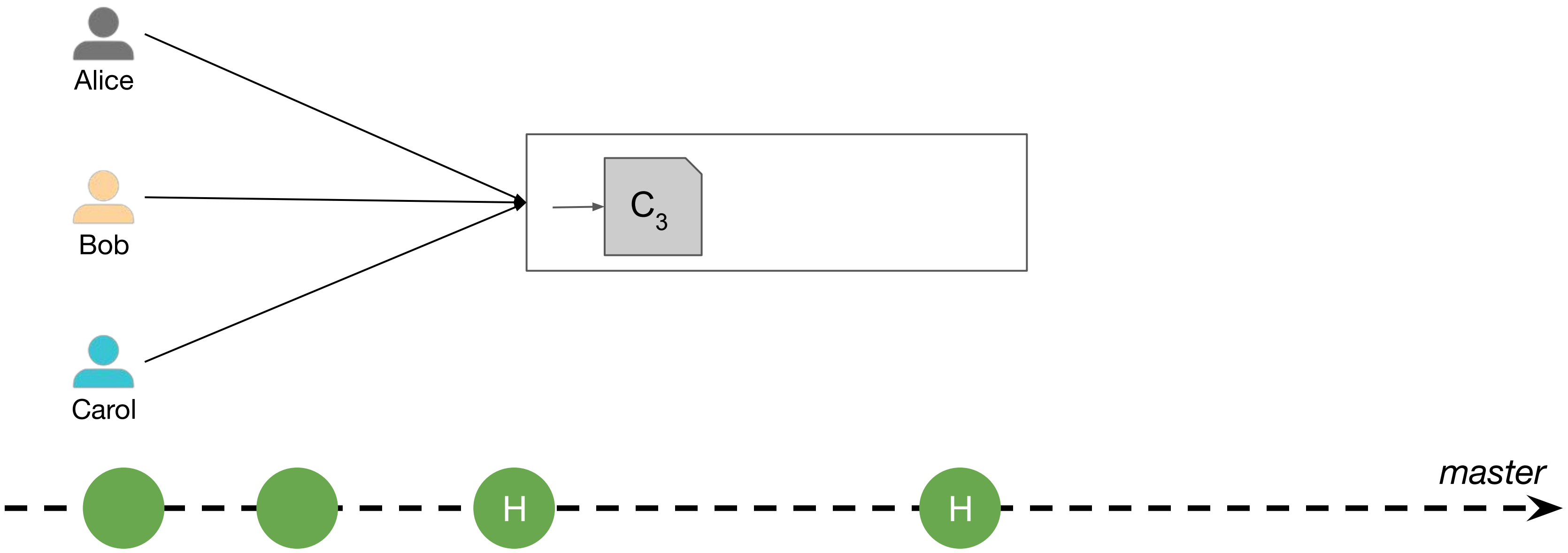
Keeping master green: Queue



Build steps for H ⊕ C_2 fails and C_2 is rejected.

UBER

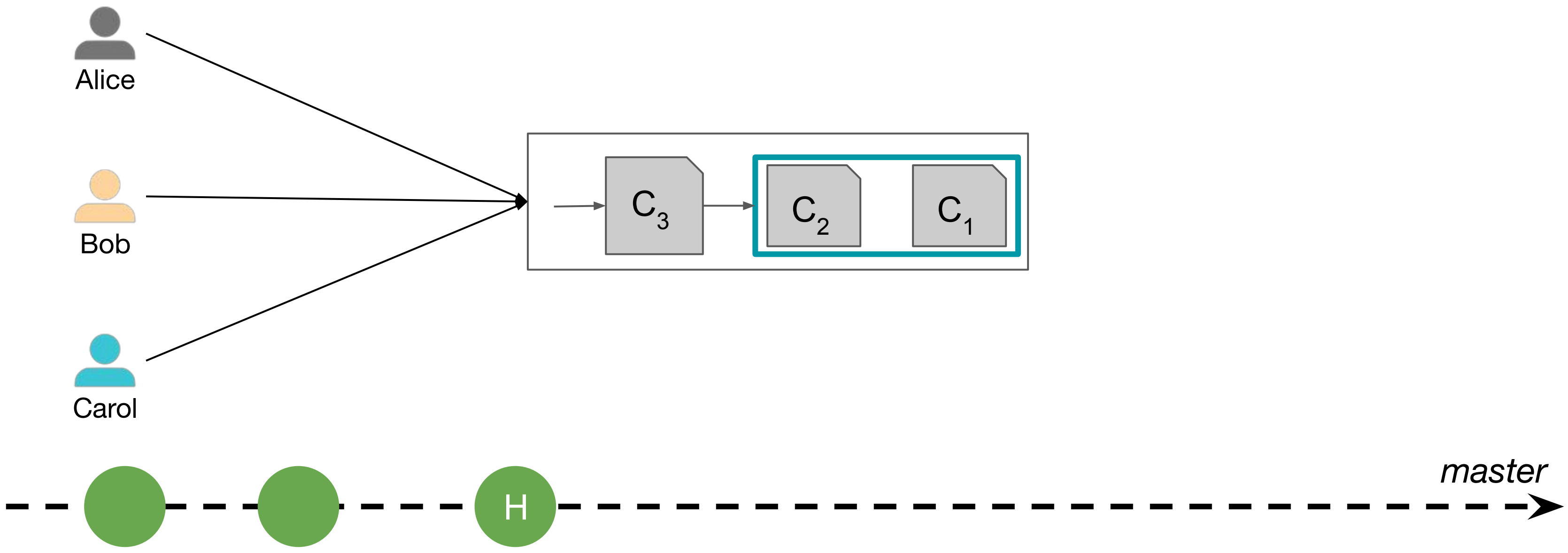
Keeping master green: Queue



✅ Guarantees an always green master by serializing changes
❌ Does not scale to 1000s of changes/day

UBER

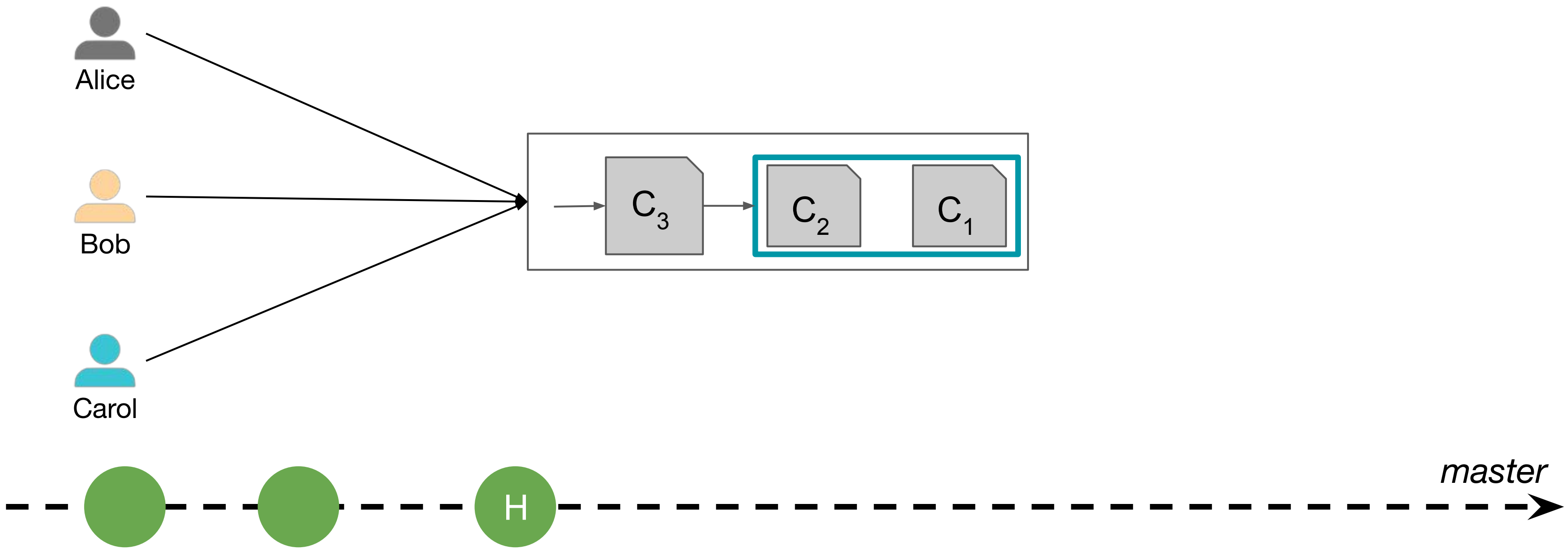
Keeping master green: Batching changes



C_1 and C_2 are batched and build steps are run.

UBER

Keeping master green: Batching changes



✔ Improves the throughput if batches succeed more often than not

✘ Testing batches masks intermediate changes that fail

✘ Batches will fail often as the size of the batch increases

What happens when batches fail?

UBER

Keeping master green: Goals

Guarantee serializability

- Illusion of a single queue when committing changes
- Git only offers serializability of patches

Provide reasonable SLAs

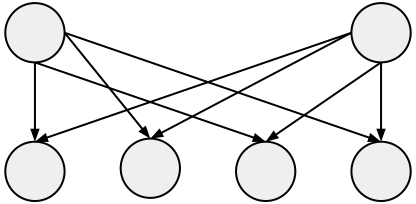
- Overheads should be short enough for developers to trade speed for correctness!

Challenge: how to do this at scale? (1000s of commits/day)

Submit Queue: Overview

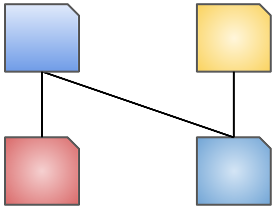
Speculation Engine

- Speculates on success/failure of changes
- Builds speculation graph



Conflict Analyzer

- Determines independent changes
- Constructs conflict graph

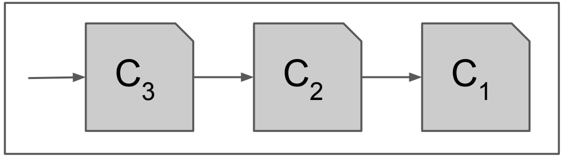


Planner Engine

- Selects most valuable builds from speculation engine
- Execute builds and commit changes

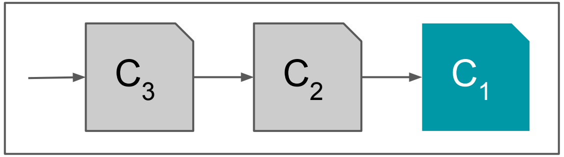
UBER

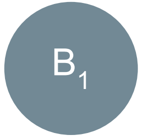
Speculation Tree



C_1, C_2, C_3 - pending changes

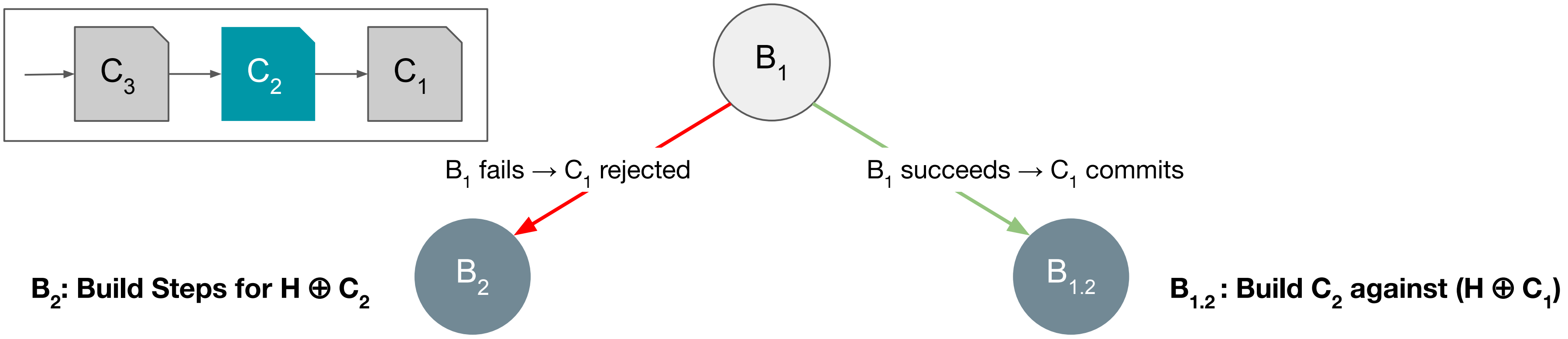
Speculation Tree





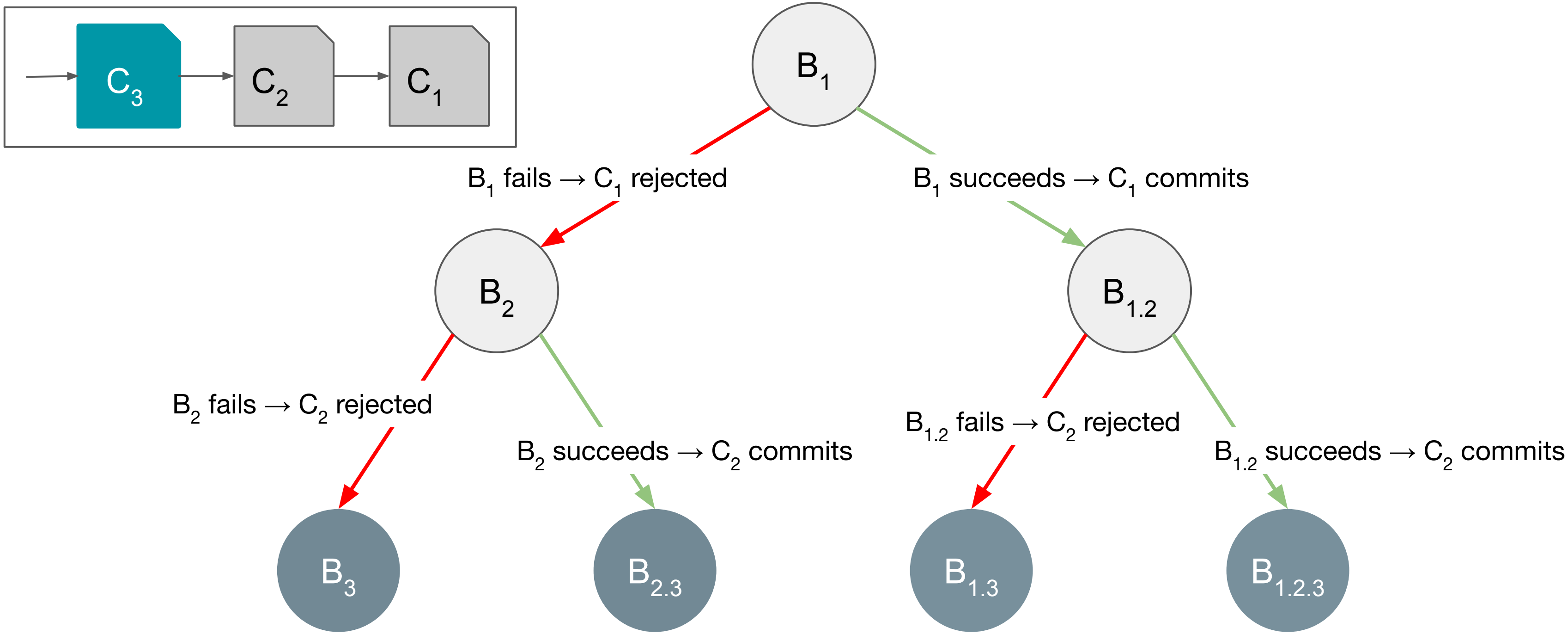
B_1: Build Steps for H ⊕ C_1

Speculation Tree



1. Precompute the outcome of committing C_2 under different realities
2. Commit or reject C_2 based on the outcome of B_1 and one of $\{B_2, B_{1.2}\}$

Speculation Tree



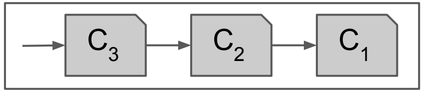
Challenge: Which builds to run?

UBER

Approach #1: Speculate Them All

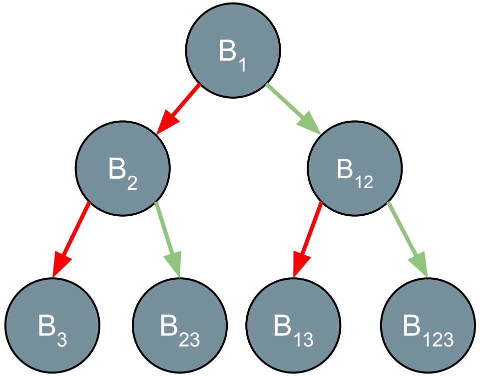
Speculate on all possible outcomes equally

- Selects builds in a breadth-first order



Does not scale for 1000s of changes/day

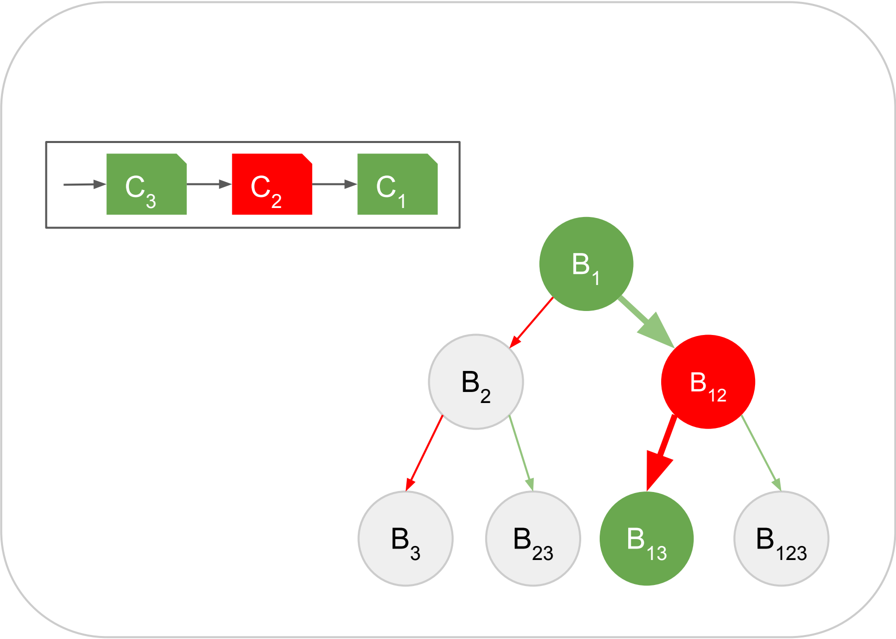
- Need to run 2^n builds in parallel to commit *‘n’* changes



Leads to substantial waste of resources

UBER

Speculate Them All: Resource Wastage



UBER

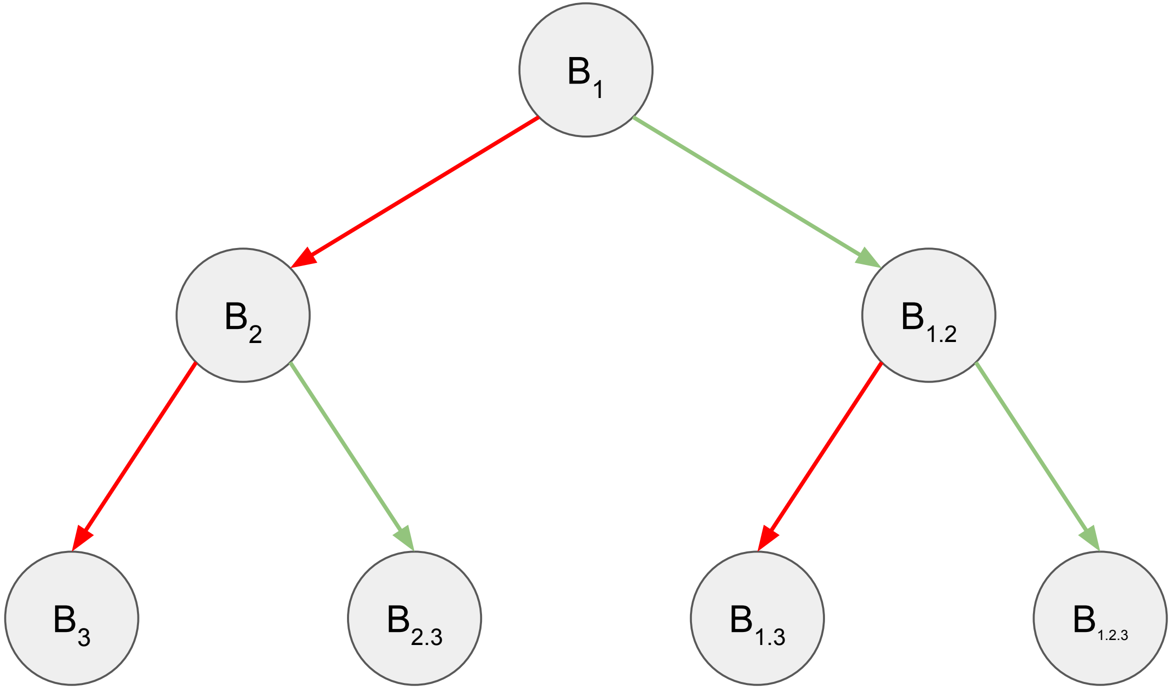
Speculate Them All: Observation

If we select and execute builds whose *outcomes* are **most likely to be needed**, then we require only **n** (out of 2^n) **builds**.

Challenge: Which *‘n’* builds are likely to be needed?

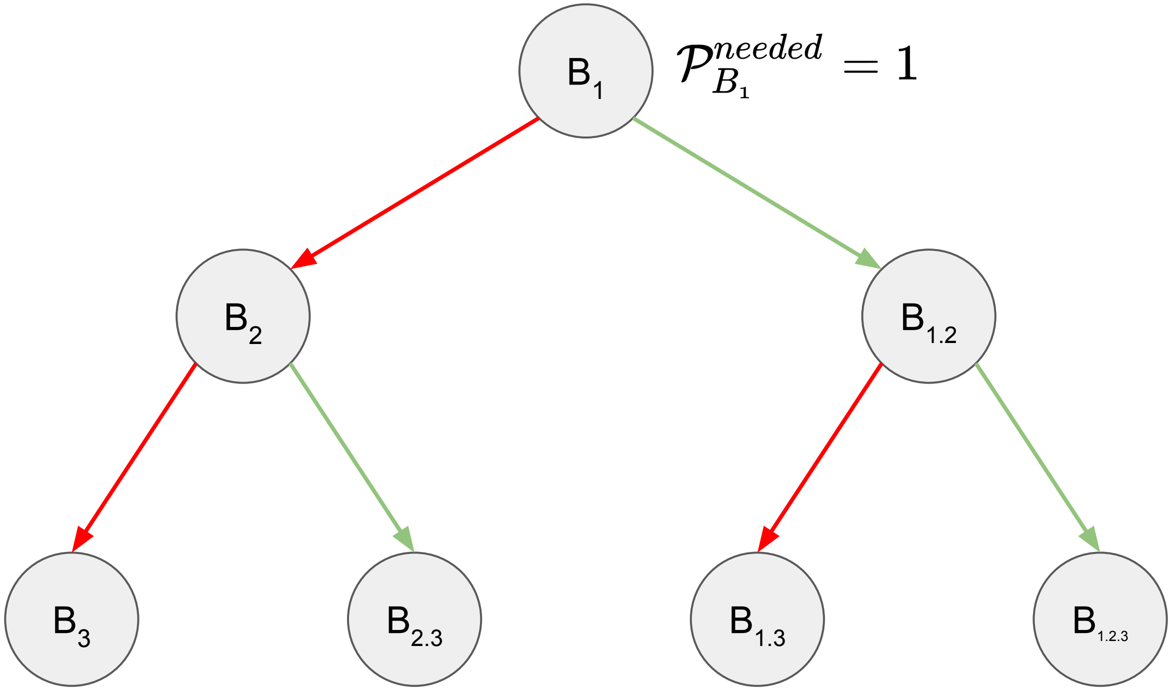
UBER

Probabilistic Speculation



$\mathcal{P}_{B_{\mathcal{C}}}^{needed}$ represents the prob. the result of the build B_C is used to make to commit/reject C.

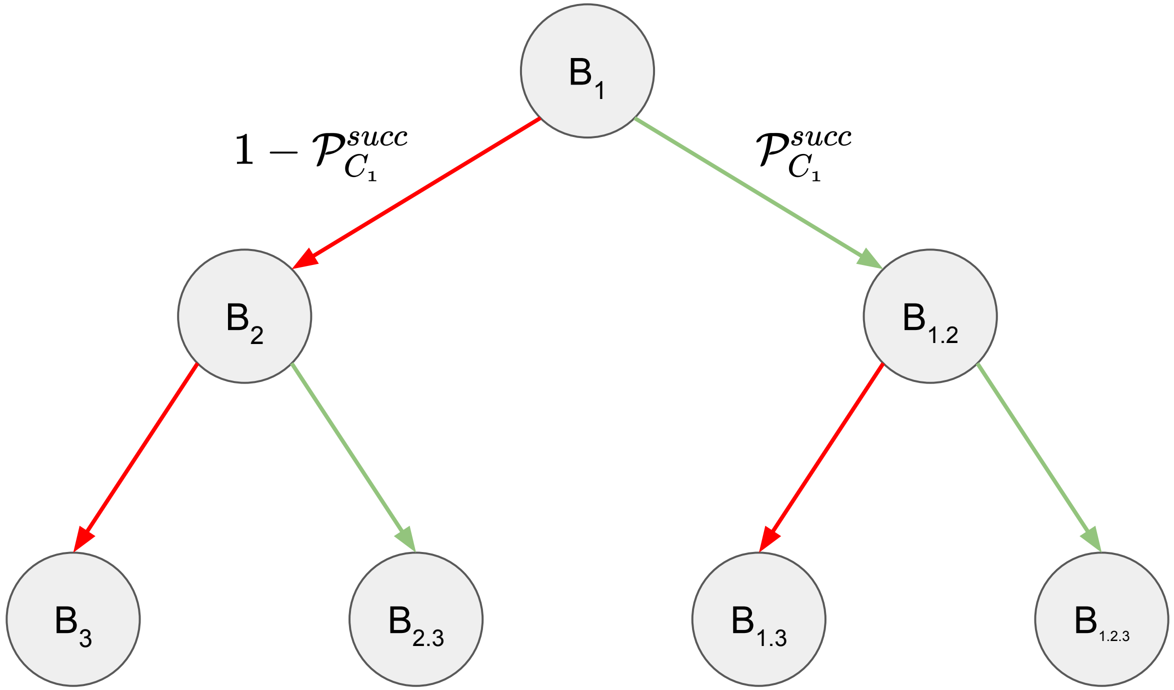
Probabilistic Speculation



Root B_1 is always needed as is used to determine if C_1 can be committed

UBER

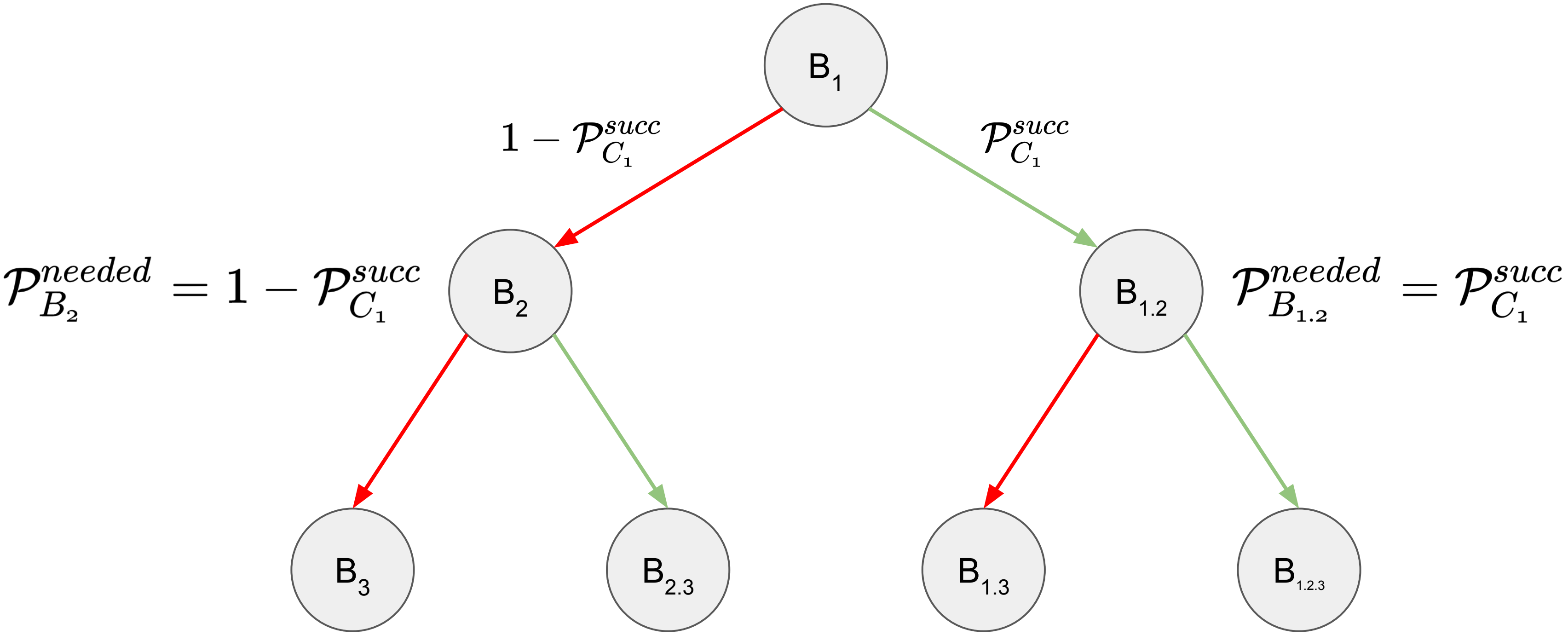
Probabilistic Speculation



$\mathcal{P}_{C_1}^{succ}$ represents the prob. that change C_1 succeeds individually

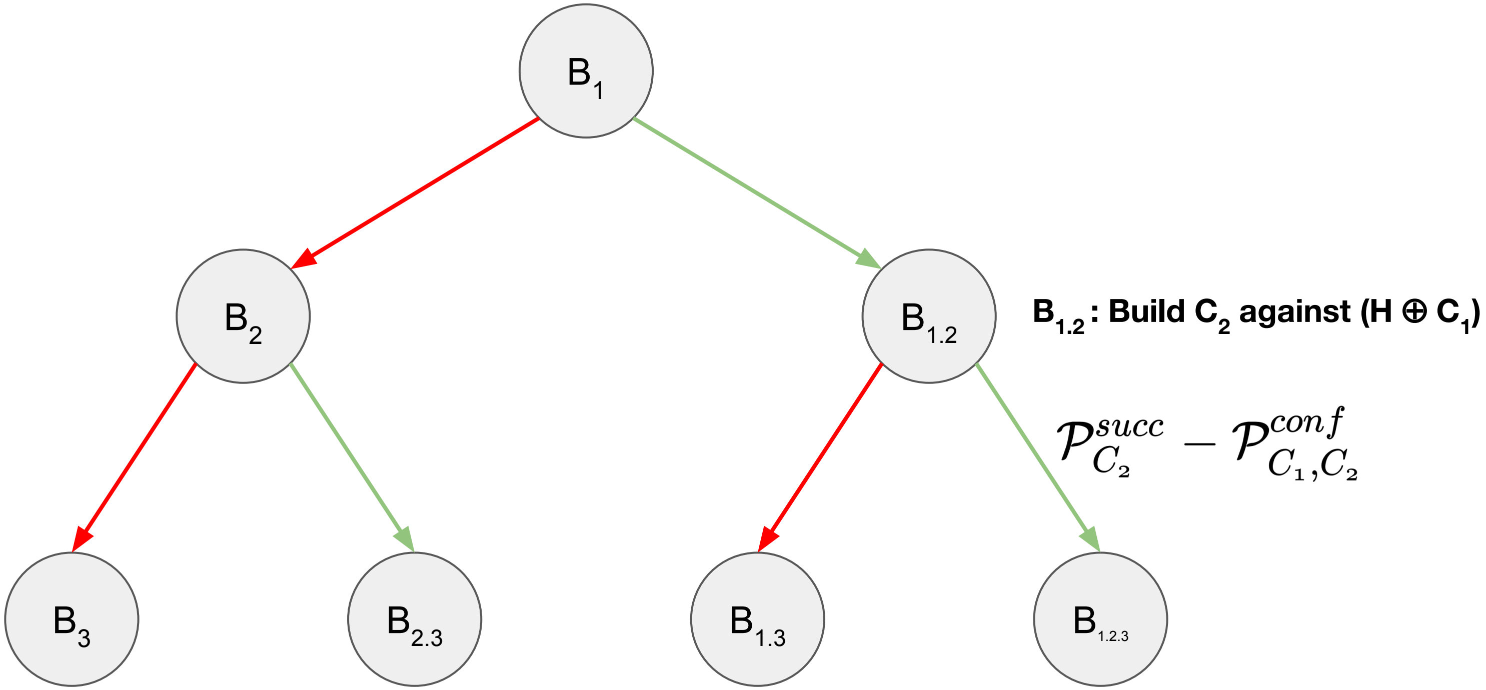
UBER

Probabilistic Speculation



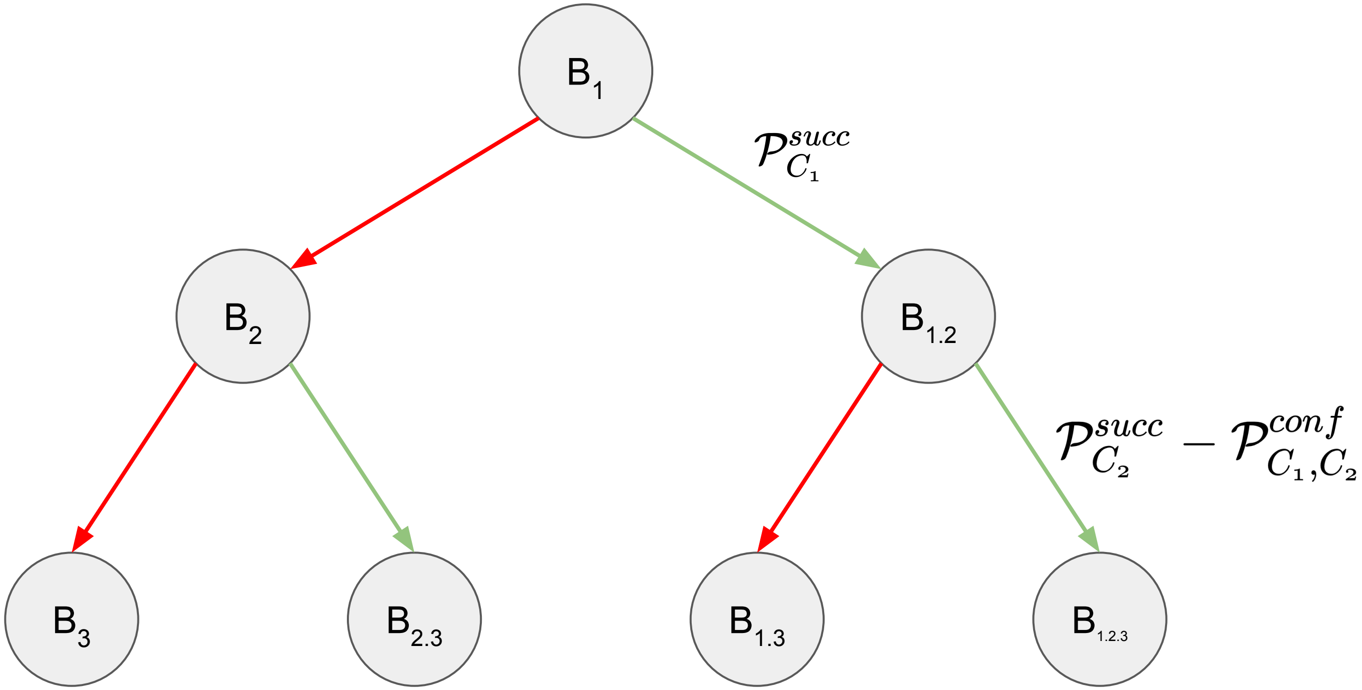
$\mathcal{P}_{C_1}^{succ}$ represents the prob. that change C_1 succeeds individually

Probabilistic Speculation



$\mathcal{P}^{conf}_{C_1,C_2}$ represents the prob. that C_2 conflicts with C_1

Probabilistic Speculation

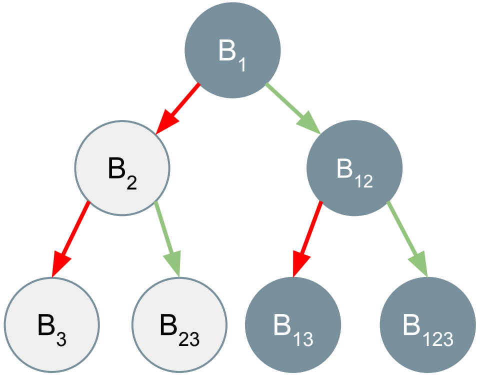


$$\mathcal{P}^{needed}_{B_{1.2.3}} = \mathcal{P}^{succ}_{C_1} \cdot (\mathcal{P}^{succ}_{C_2} - \mathcal{P}^{conf}_{C_1,C_2})$$

Probabilistic Speculation: Summary

Choose ***most valuable builds*** by determining

- Probability of success of a change $\mathcal{P}_{C_i}^{succ}$
- Probability of a conflict bet. changes $\mathcal{P}_{C_i,C_j}^{conf}$



UBER

Evaluating $\mathcal{P}_{C_i}^{succ}$ and $\mathcal{P}_{C_i,C_j}^{conf}$

- **Logistic regression to train prediction models**
  - Feature set includes 100+ hand-picked features
  - Prediction accuracy of 97%

Change

- # affected targets
- # git commits
- # files changed
- status of pre-submit checks

Developer

- developer name
- employment proficiencies

Speculation

- dynamic features to re-adjust weights based on initial predictions
- # speculations succeeded
- # speculations failed

UBER

Features for Training ML Models

Change

- # affected targets
- # git commits
- # files changed
- status of pre-submit checks

Revision

- revision is a container for changes
- # changes submitted
- revert and test plans
- # Submit attempts made

Developer

- developer name
- employment proficiencies

Speculation

- dynamic features to re-adjust weights based on initial predictions
- # speculations succeeded
- # speculations failed

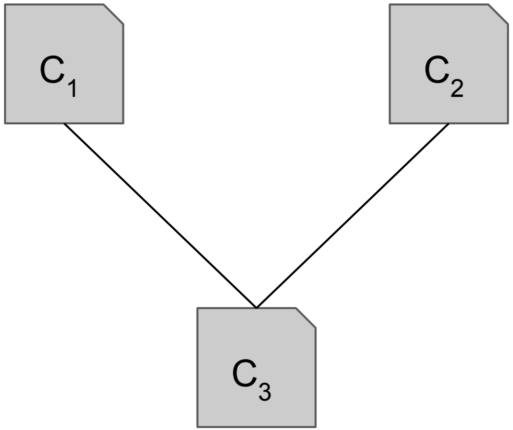
Conflict Analyzer

- **So far, we assumed all changes potentially conflict with each other**
  - Cannot commit in parallel

- **What if changes can be proved to be independent?**
  - Commit changes in parallel
  - Trim speculation space

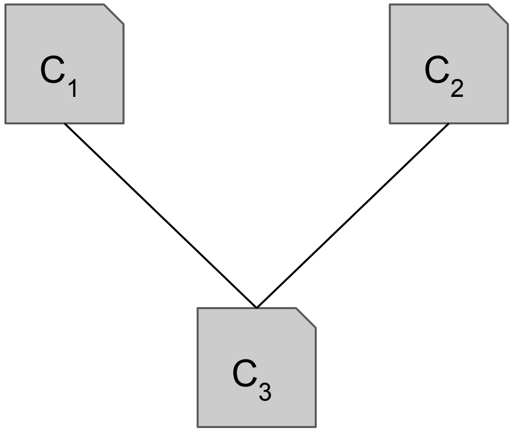
- **We use Conflict Analyzer to find independent changes**

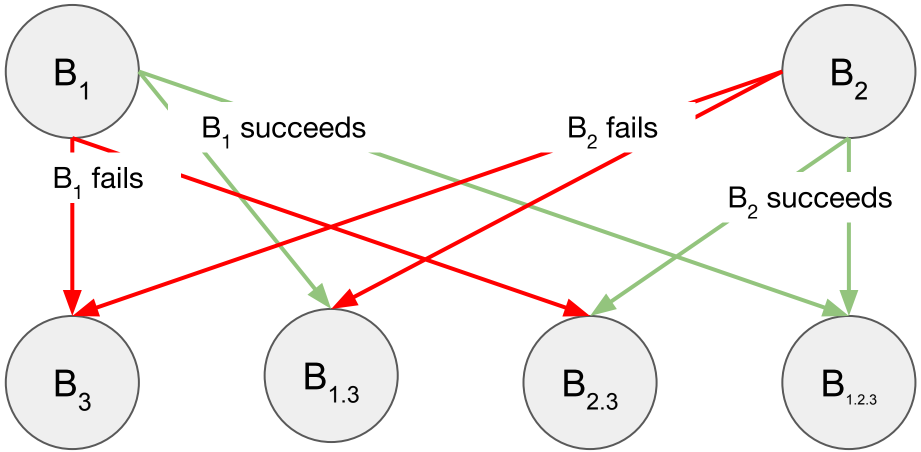
Conflict Analyzer: Commit Changes in Parallel



Conflict graph for changes C_1, C_2, C_3 where C_1 and C_2 are independent and conflict with C_3.

Conflict Analyzer: Commit Changes in Parallel

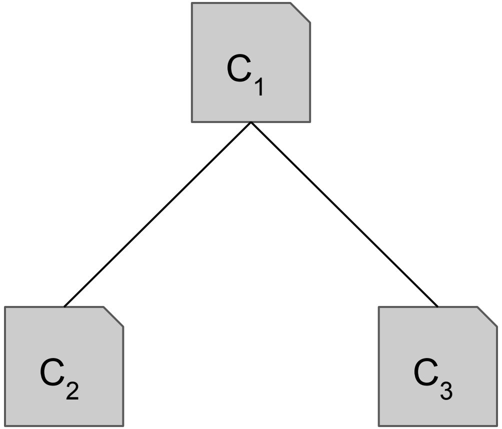




Insight: Changes C_1 and C_2 can be committed in parallel.

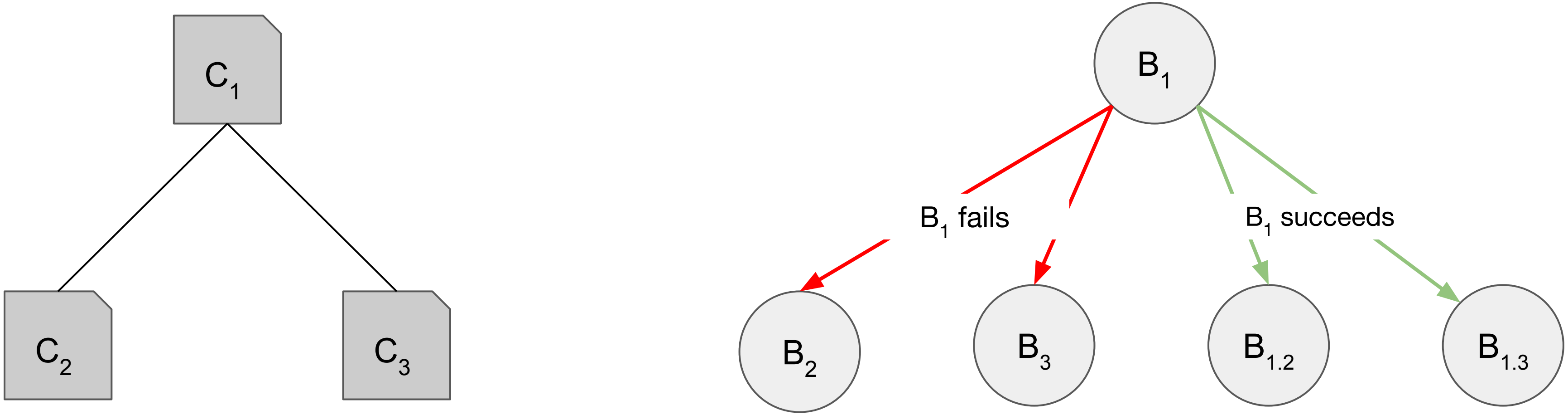
UBER

Conflict Analyzer: Trim Speculation Space



Conflict graph for C_1, C_2, C_3 where C_1 conflicts with independent changes C_2 and C_3.

Conflict Analyzer: Trim Speculation Space

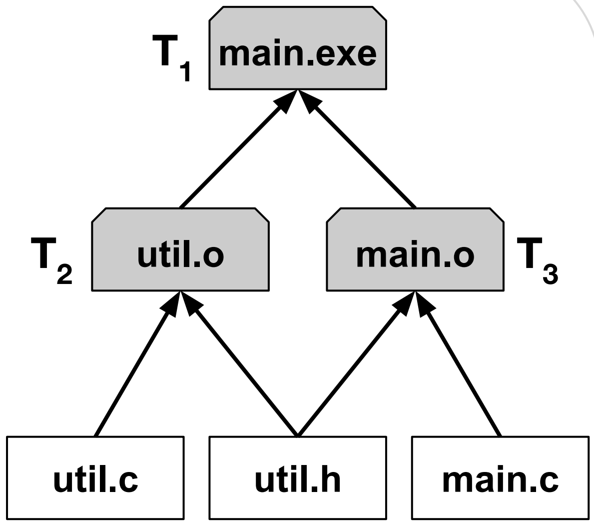


Insight: Because C_3 does not speculate on C_2, # of possible builds for C_3 reduces to 2.

UBER

Conflict Analyzer: Detecting conflicts at scale

- **Build system** to detect if changes are independent
- Code partitioned into smaller entities called *targets*
- Every change affects a set of *targets*

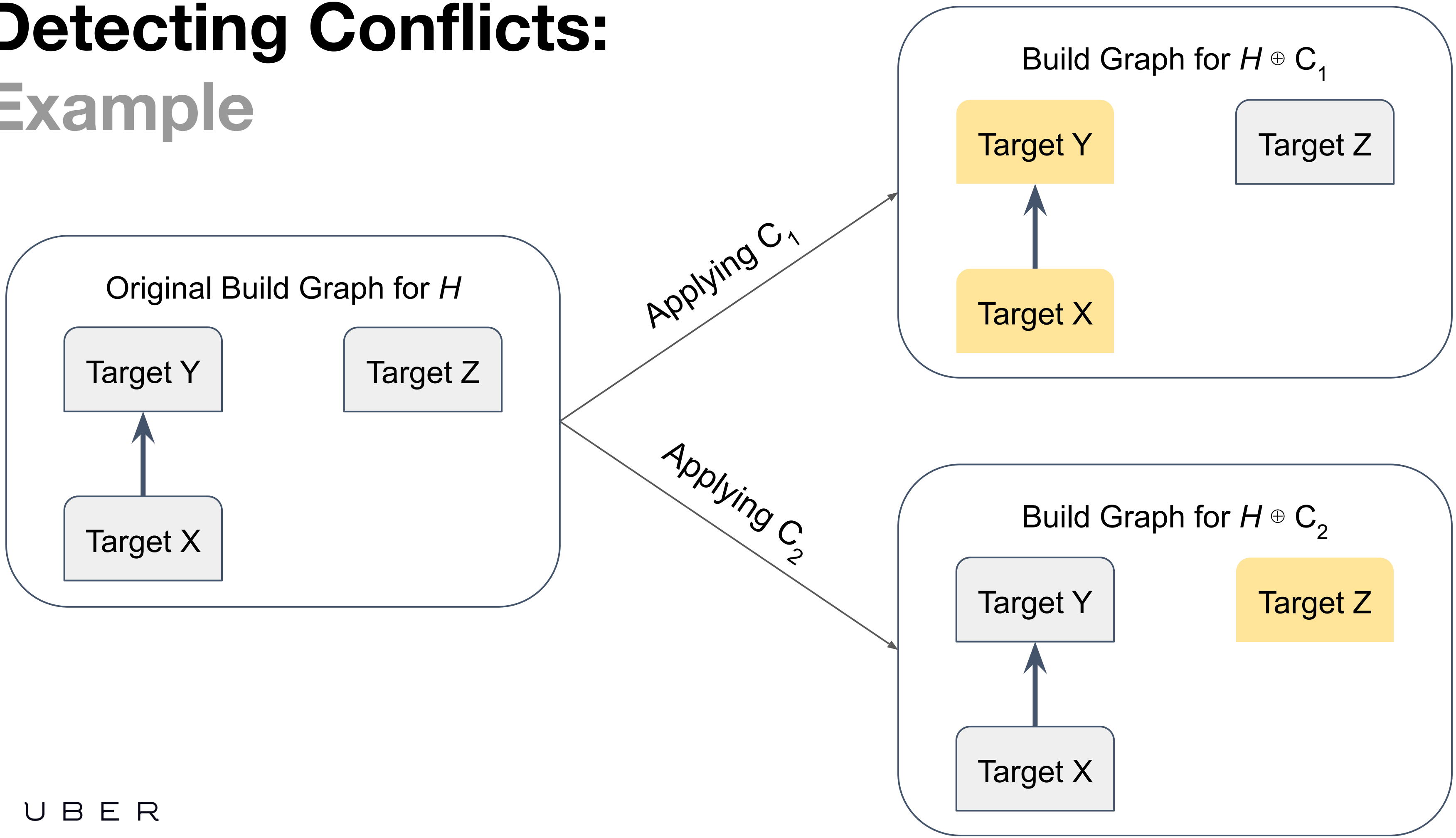


Example build graph

Detecting Conflicts: Intuition

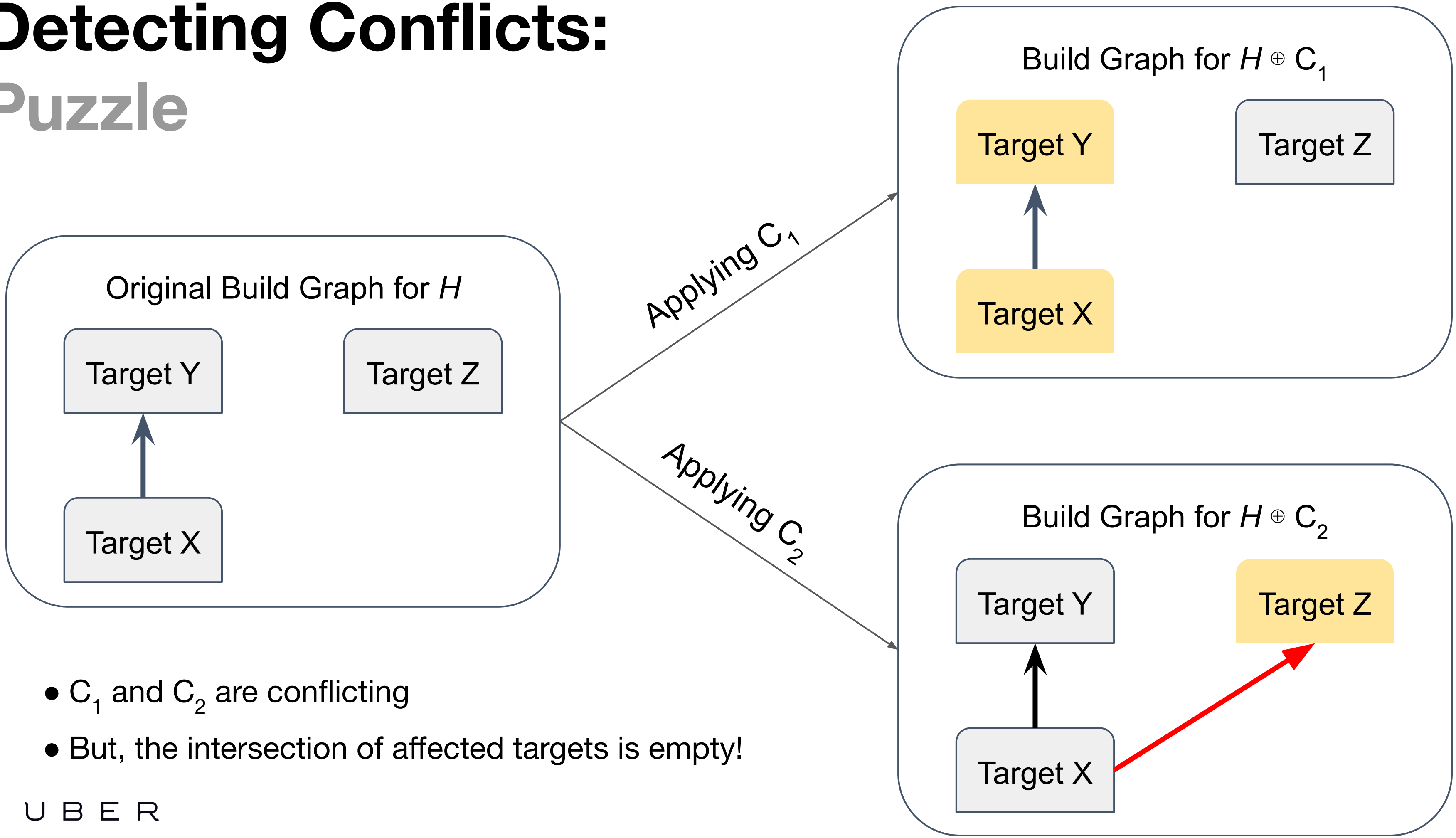
Two changes are independent if they affect a disjoint set of targets.

Detecting Conflicts: Example



UBER

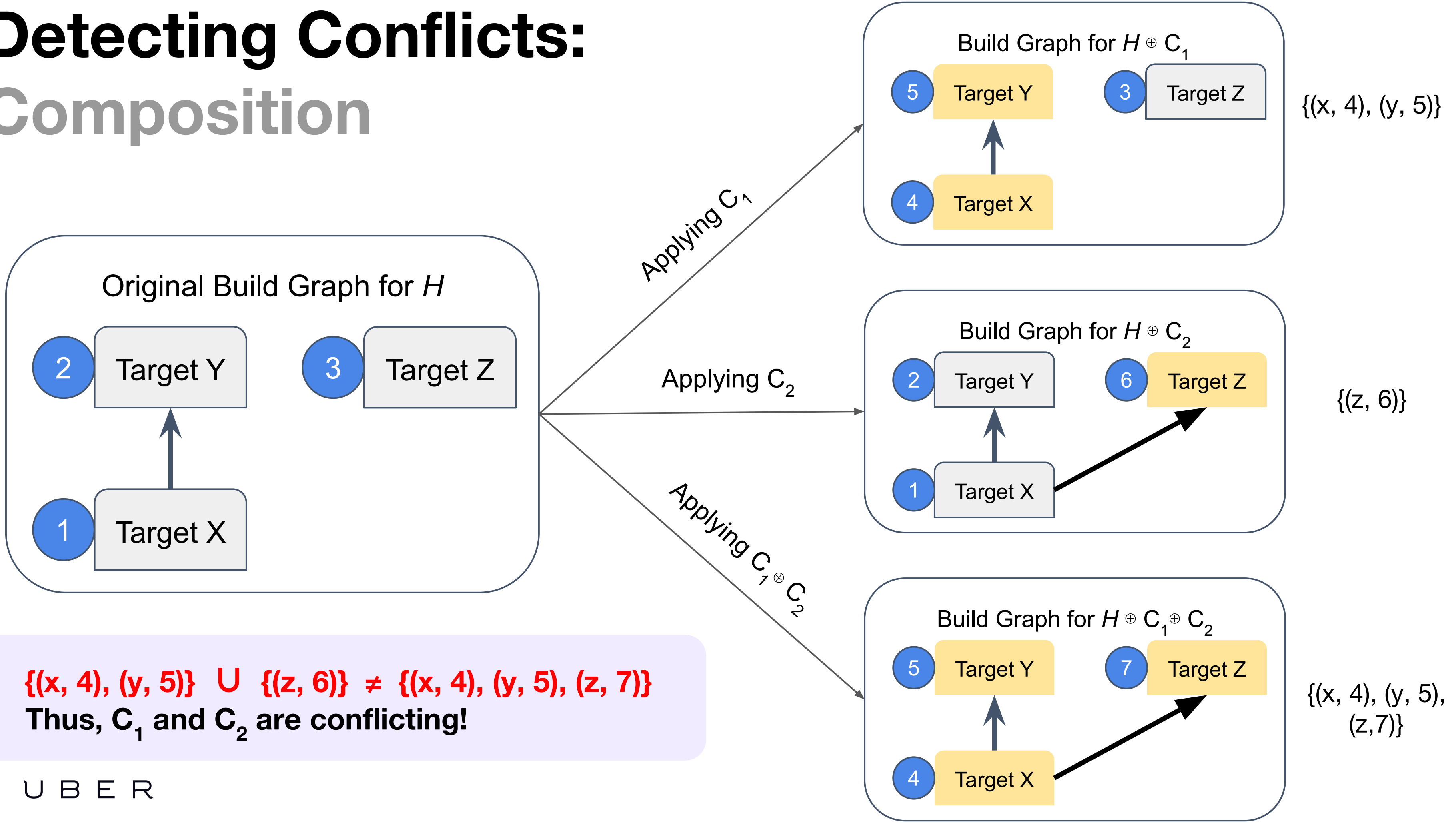
Detecting Conflicts: Puzzle



- C_1 and C_2 are conflicting
- But, the intersection of affected targets is empty!

UBER

Detecting Conflicts:
Composition



{(x, 4), (y, 5)} ∪ {(z, 6)} ≠ {(x, 4), (y, 5), (z, 7)}
Thus, C_1 and C_2 are conflicting!

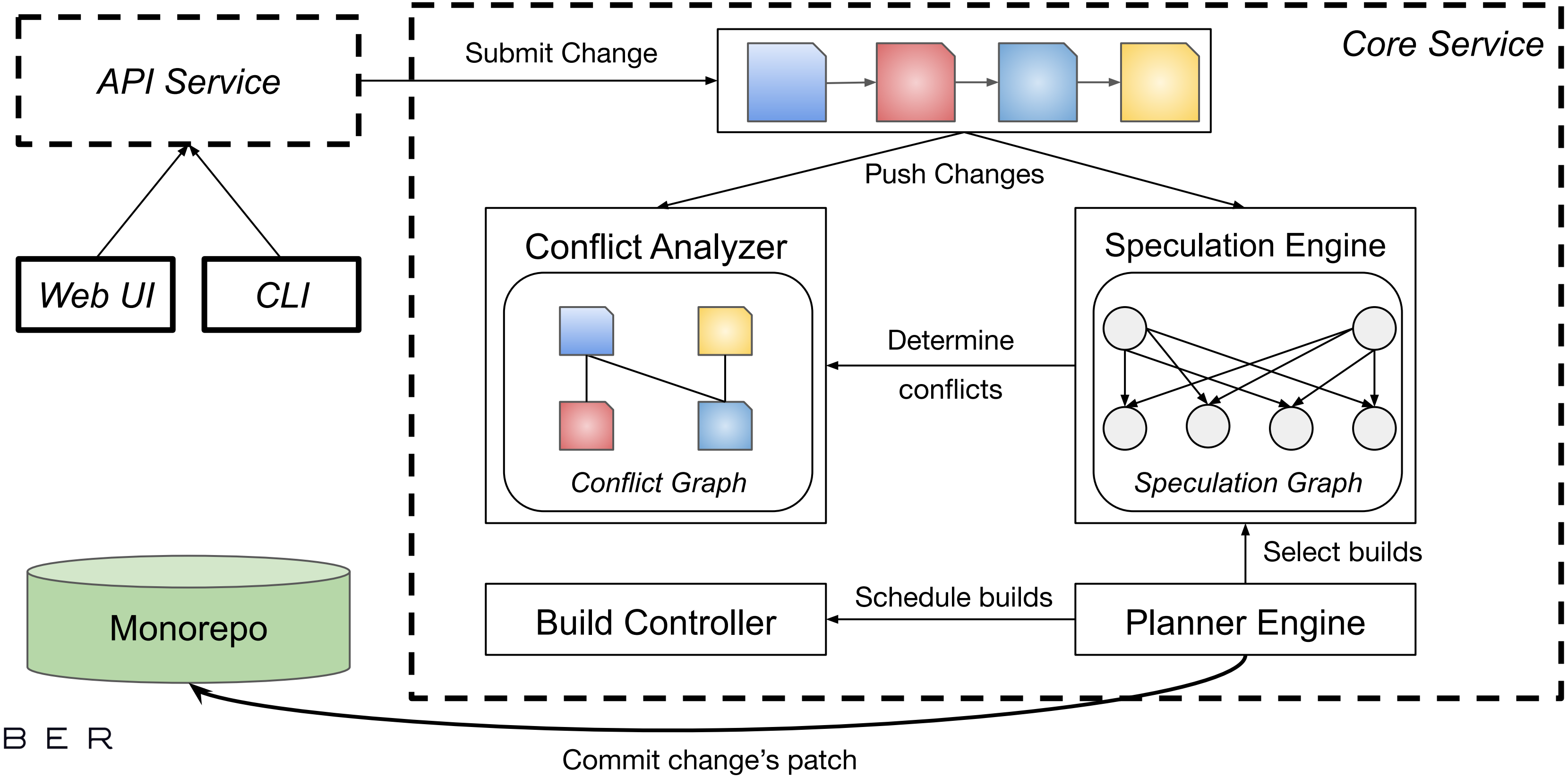
UBER

Detecting Conflicts: Summary

- **Intersection Approach**
  ❌ Does not detect all kinds of conflicts
- **Union Approach**
  ❌ Determining conflicts for n changes requires n^2 build graphs!
- **Hybrid Approach**
  ✅ Only **7.9%** of changes cause a change to the build graph
- Union Graph Approach (details in paper)

UBER

Submit Queue: Architecture Overview



Evaluation

Questions

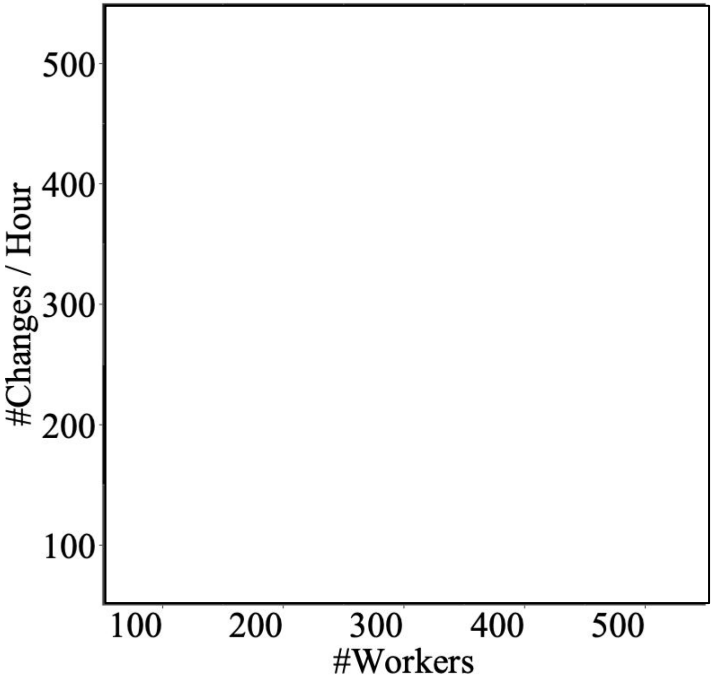
- How does Submit Queue performance compare against other strategies?
  - Queue, Speculate-all, Optimistic

- What is the impact of conflict analyzer?

Setup

- Implemented an Oracle that predicts outcome of a change correctly
  - All results normalized against the Oracle
- Ingested real changes into our system at different rates

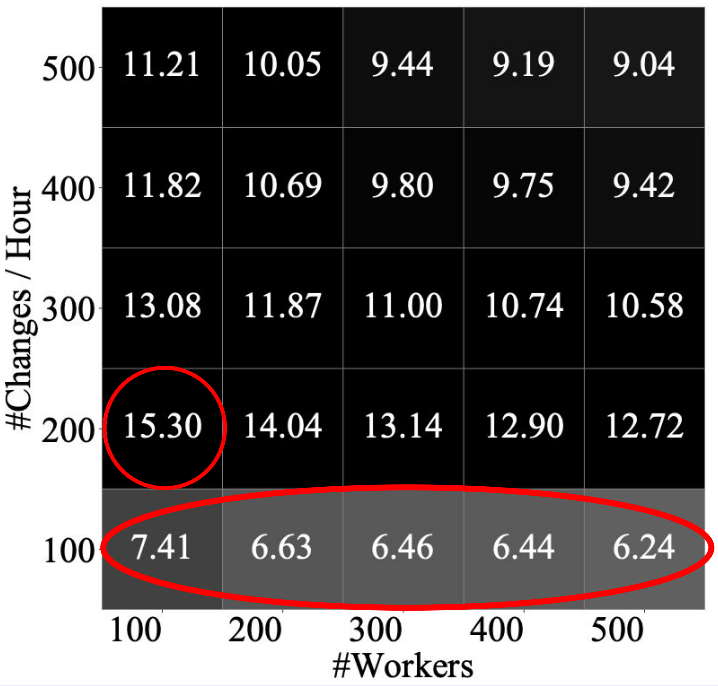
Evaluation: Submit Queue Performance



UBER

Evaluation: Submit Queue Performance (P50)

Speculate-all



Speculate-all suffers up to **15x** slowdown compared to the Oracle.

UBER

Evaluation: Submit Queue Performance (P50)

Speculate-all

#Changes / Hour \ #Workers	100	200	300	400	500
500	11.21	10.05	9.44	9.19	9.04
400	11.82	10.69	9.80	9.75	9.42
300	13.08	11.87	11.00	10.74	10.58
200	15.30	14.04	13.14	12.90	12.72
100	7.41	6.63	6.46	6.44	6.24

Optimistic speculation

#Changes / Hour \ #Workers	100	200	300	400	500
500	8.54	8.72	8.62	8.57	8.77
400	8.75	8.70	8.67	8.74	8.69
300	7.33	7.63	7.64	7.56	7.65
200	9.60	9.62	9.62	9.64	9.64
100	7.46	7.46	7.44	7.44	7.44

Optimistic speculation performs better than speculate-all esp. under contention.

Evaluation: Submit Queue Performance (P50)

Speculate-all

#Changes / Hour \ #Workers	100	200	300	400	500
500	11.21	10.05	9.44	9.19	9.04
400	11.82	10.69	9.80	9.75	9.42
300	13.08	11.87	11.00	10.74	10.58
200	15.30	14.04	13.14	12.90	12.72
100	7.41	6.63	6.46	6.44	6.24

Optimistic speculation

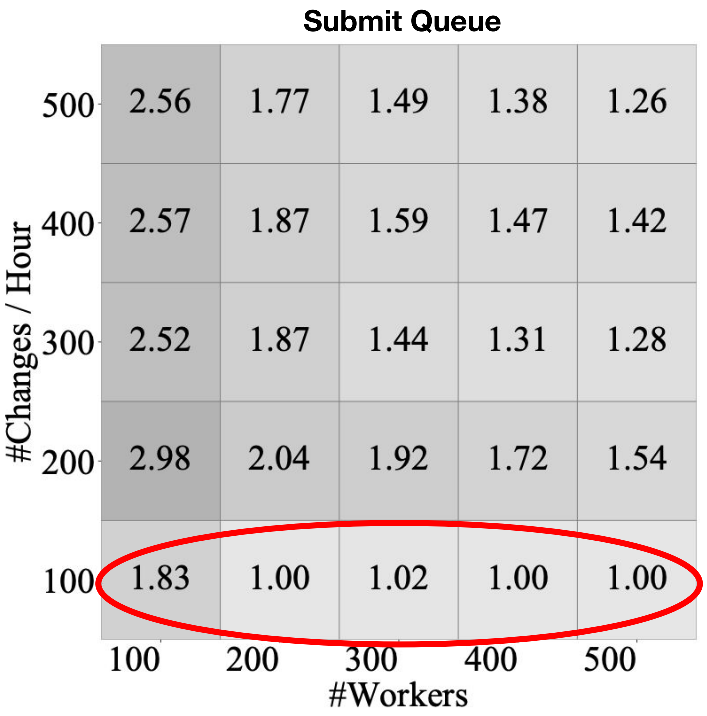
#Changes / Hour \ #Workers	100	200	300	400	500
500	8.54	8.72	8.62	8.57	8.77
400	8.75	8.70	8.67	8.74	8.69
300	7.33	7.63	7.64	7.56	7.65
200	9.60	9.62	9.62	9.64	9.64
100	7.46	7.46	7.44	7.44	7.44

Submit Queue

#Changes / Hour \ #Workers	100	200	300	400	500
500	2.56	1.77	1.49	1.38	1.26
400	2.57	1.87	1.59	1.47	1.42
300	2.52	1.87	1.44	1.31	1.28
200	2.98	2.04	1.92	1.72	1.54
100	1.83	1.00	1.02	1.00	1.00

Submit Queue has the best performance among all the approaches.

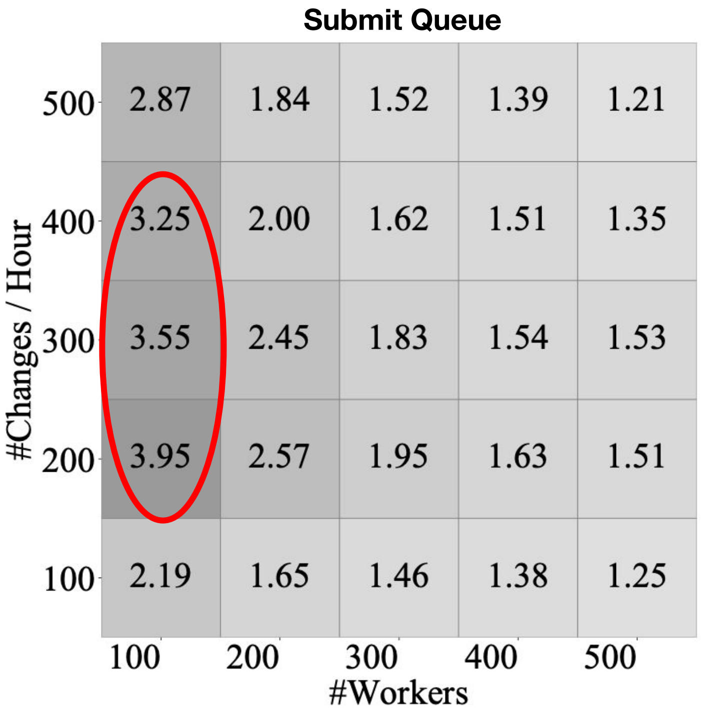
Evaluation: Submit Queue Performance (P50)



Performance matches Oracle's performance under low contention

UBER

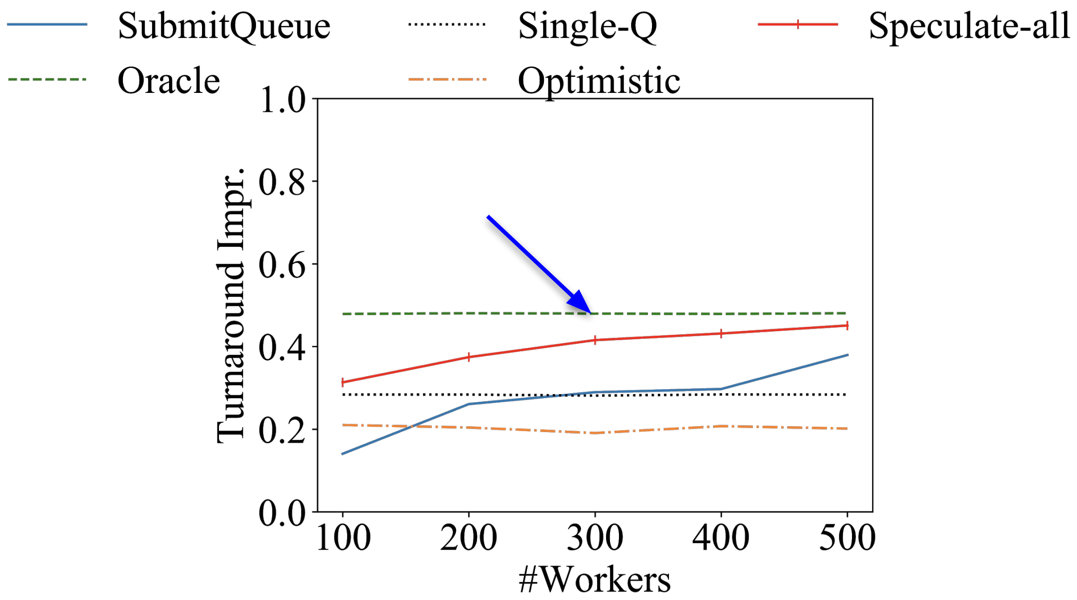
Evaluation: Submit Queue Performance (P99)



P99 turnaround time is only 4x worse under *extreme contention.*
We don't operate there typically in production.

UBER

Evaluation: Impact of Conflict Analyzer



P95 Turnaround Time Impr. for 500 changes/hour

- Oracle's turnaround time improves by up to 50% with conflict analyzer.
- Benefit for SQ and Speculate-all steadily converges towards Oracle.

UBER

Submit Queue guarantees always-green master

- **Probabilistic speculation** powered by *logistic regression* to select builds that are likely to succeed, and execute them in parallel

- **Conflict analyzer** to commit independent changes in parallel, and trim the speculation space.

- **Evaluated** Submit Queue in production deployment

Thank you!