Continuous Integration and Continuous Deployment

Pragadeesh and Yue

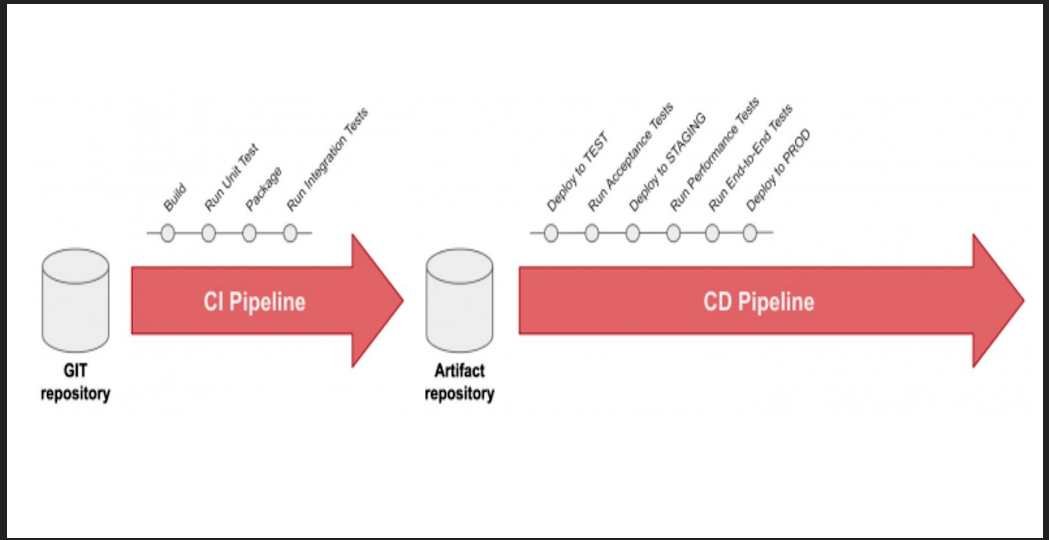
Summary

- Overview of CI/CD -Yue
- Why CI/CD important? -Prag
- CI/CD for microservice -Prag
- CD and Facebook/OANDA case studies -Prag
- CI and Uber SubmitQueue pipeline -Yue
- CI/CD in monolithic and microservice -Yue
- Q/A -Yue, Prag

Overview of CI/CD

Continuous Integration - regularly build, test, and merge code changes into main branch.

Continuous Deployment - automatically test and release changes from the repo to production.



Overview CI/CD

- **CI** is generally a standard across all software projects.
- **CD** on the other hand is not. Example: Aerospace Industries, Healthcare, etc.

Why is CI/CD useful?

- Release software with less risks
  - CI/CD takes care of the automated testing, deployment and rollbacks.
- Can improve developer productivity.
  - More automation, less context switching, etc
- Ship features and fix bugs faster.
- Ability to push out small changes and iterate on them continually.

CI/CD for microservices

- Requirements
  - Highly cohesive and loosely coupled services and teams
  - Teams and engineers that can make key-decisions independently

CI/CD for microservices

- Challenges:
  - Tools to support deployment and experience in managing them.
  - Senior Management buy-in.
  - Investment on the tools

Despite the challenges, a lot of companies use CI/CD to manage their release.

CI/CD in practice

Steps involved:

- Code Review
- Testing
  - Unit tests, integration tests, performance tests, etc
- Release engineering
  - Assess the risk and manage the deployment
- Deployment

Continuous Deployment

- Blue-green deployment
  - Deploy to a small fraction of people and dial it up
- Dark launches
  - Launch during non-peak hours
- Staging
  - Test the builds in multiple staging environments
  - Simulate some traffic

Transition to CI/CD

- Automated Testing infrastructure
  - Unit tests, Integration tests, Shadow tests, performance tests, etc

- Deployment Management System
  - Code reviews, VCS, deployment scheduling, staging pipelines, Rollbacks

CD Case Studies At Facebook and OANDA

- OANDA
  - Currency trading system that manages trades worth many billions every day
  - Small team with about 100 engineers
  - Code check-in -> Deployment Pipelines -> ad-hoc alert system using emails
- Facebook
  - Billions of queries per second
  - 1000s of engineers
  - Code check-in -> Release engineering -> Error reporting (SEV)

Observations

- Productivity scaled with the size of the engineering organisation. (20x developers over a span of 6 years)

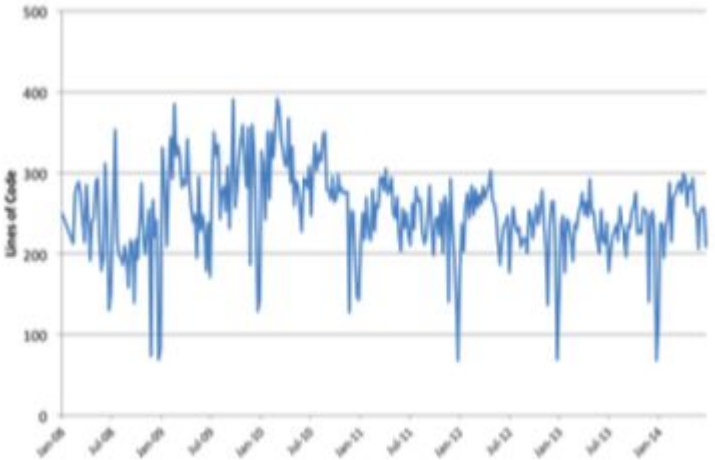


Figure 1: Lines of modified or added code deployed per developer per week at Facebook.

Observations

- Productivity scaled as product matured, became larger and more complex (Code base - 50x)

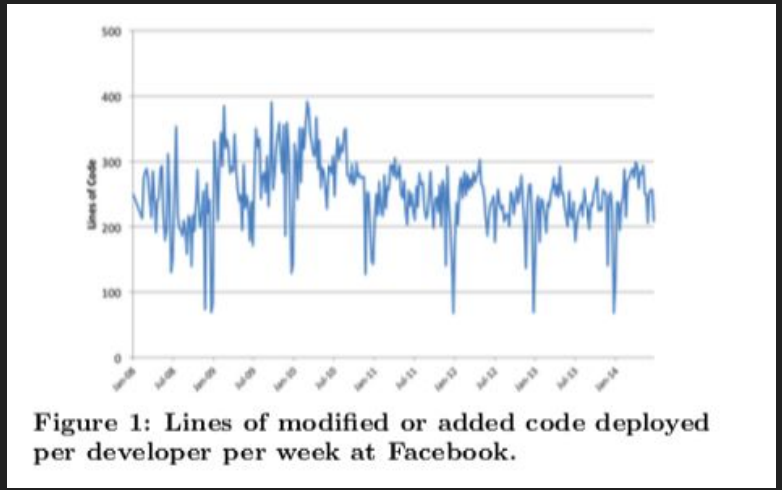


Figure 1: Lines of modified or added code deployed per developer per week at Facebook.

Observations

- The number of critical issues arising from deployments was almost constant regardless of the number of deployments.

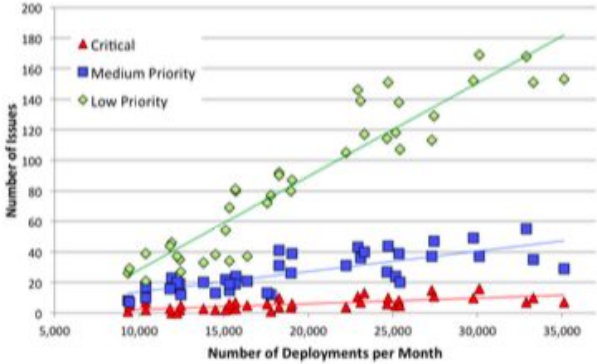


Figure 2: Number of production issues per month as a function of the number of deployments at Facebook.

Observations

- Management support can affect the productivity of an engineering organization.

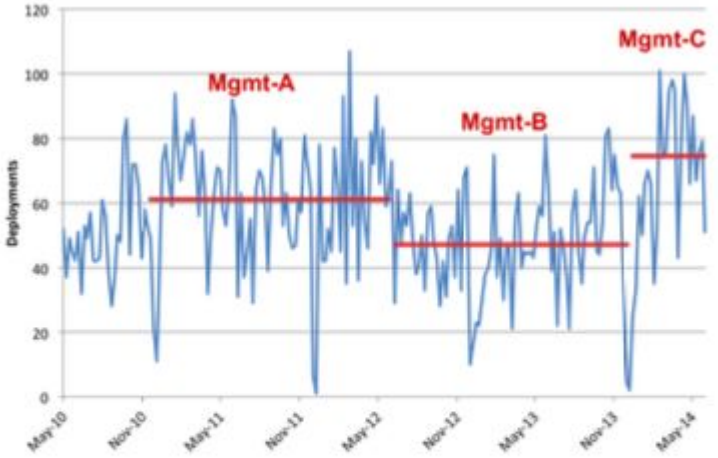


Figure 3: Number of deployments at OANDA per week over three management regimes. The red lines depict the average over the time periods shown.

Observations

- Developers prefer faster deployments over slower ones.

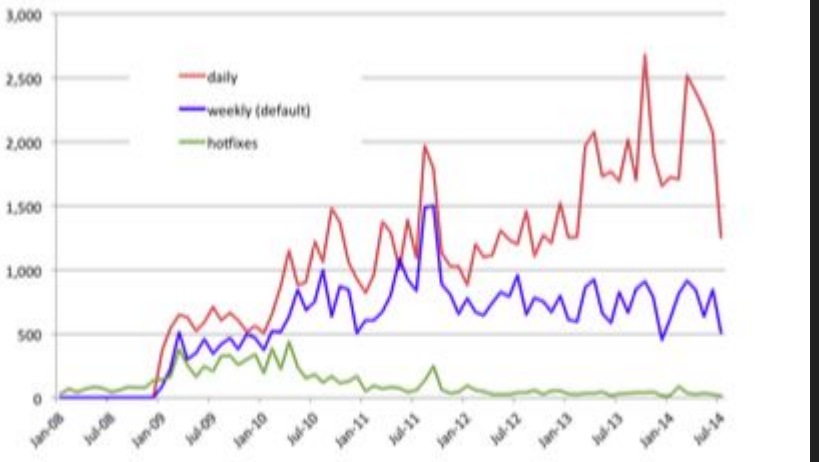


Figure 4: The number of deployment requests per week by type of request.

Lessons learnt

- Desirability of CD
  - Developers can ship faster and feel motivated by staying relevant.
- Considerable and continuous investment is required
  - 5% of FB workforce is devoted to managing deployments

Lessons learnt

- Versatile and skilled developers are required
  - Developers have to often make key decisions about the deployment
  - Open culture when it comes to code sharing
  - Manage risk vs reward

Lessons learnt

- Management buy-in
  - Empowering the developers to make the key decisions.
  - Trust the process
    - Object retrospectives when failures occur.
  - Facilitate better inter-team communication

Potential Pitfalls

- Teams avoid making bolder moves
- Suboptimal features are released because they can be improved iteratively
- Reinventing the wheel because of less communication and you have more control
- Variability in quality

CI/CD at Microsoft

- Canary testing
  - Route a small part of the production traffic to the new build to detect potential failures
- Istio
  - For advanced routing rules
  - Observability - traces, logs, etc
  - Chaos testing - Fault injections (delays, faults, etc)

CI/CD at Microsoft

- Brigade for CI/CD
  - Allows you to define event driven pipelines on JS

```
events.on("push", async () => {
  var compileStep = new Job("compile", "example/compiler:latest")
  var testStep = new Job("test", "example/tester:latest")
  var tagStep = new Job("tag:, "example/releasetagger:latest")
  // We could continue on creating the remaining steps

  await compileStep.run()
  await testStep.run()
  await tagStep.run()
  // We could continue running the remaining steps
});
```

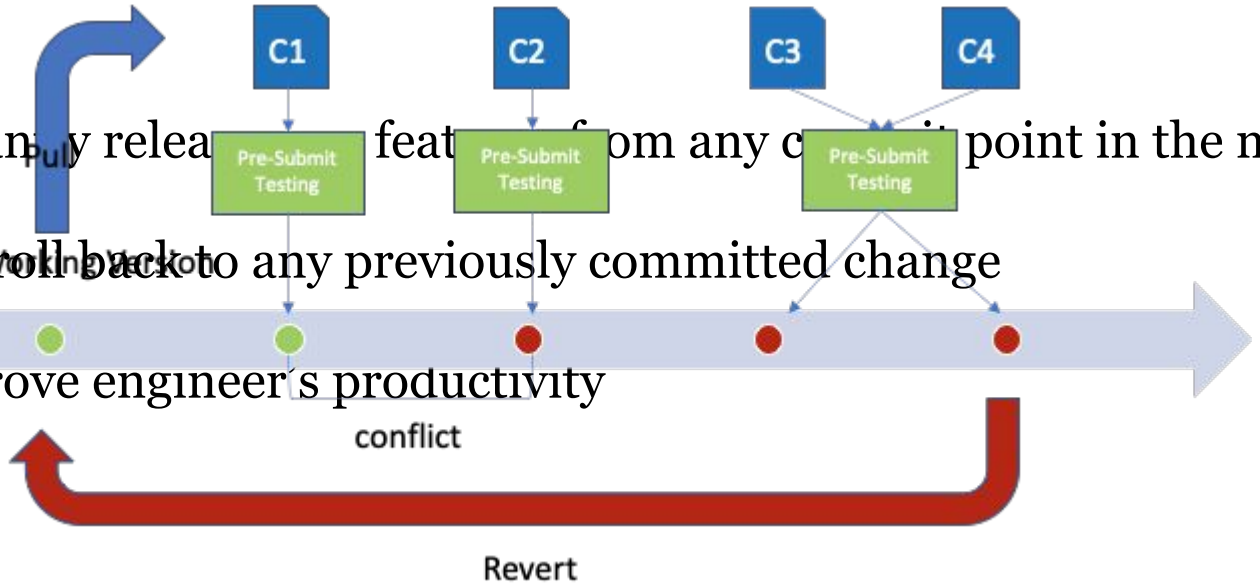
CD at Microsoft

- Brigade features:
  - Store secrets like API keys in config
  - Parallel builds
  - Trigger workflow from GitHub, Docker, etc
- Kashti
  - Companion dashboard for Brigade
  - Build logs, etc

SubmitQueue: CI Pipeline at Uber

Goal: Using CI to Keep the Master Branch Green

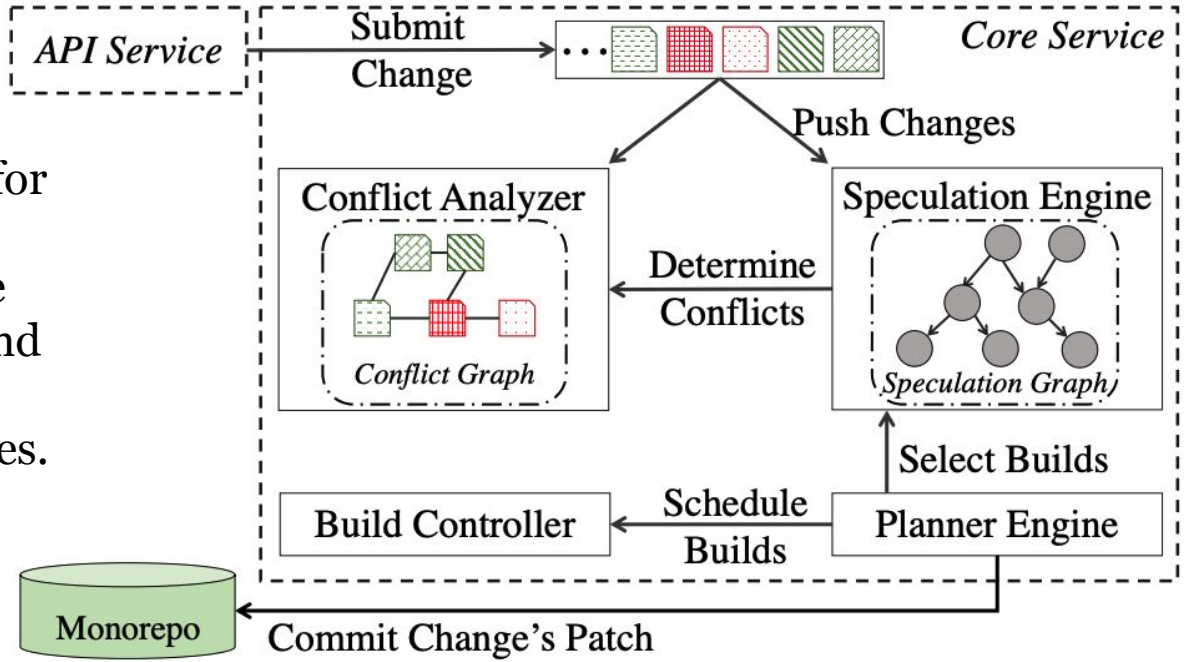
- Instantly release features from any commit point in the mainline
- Can roll back to any previously committed change
- Improve engineer's productivity



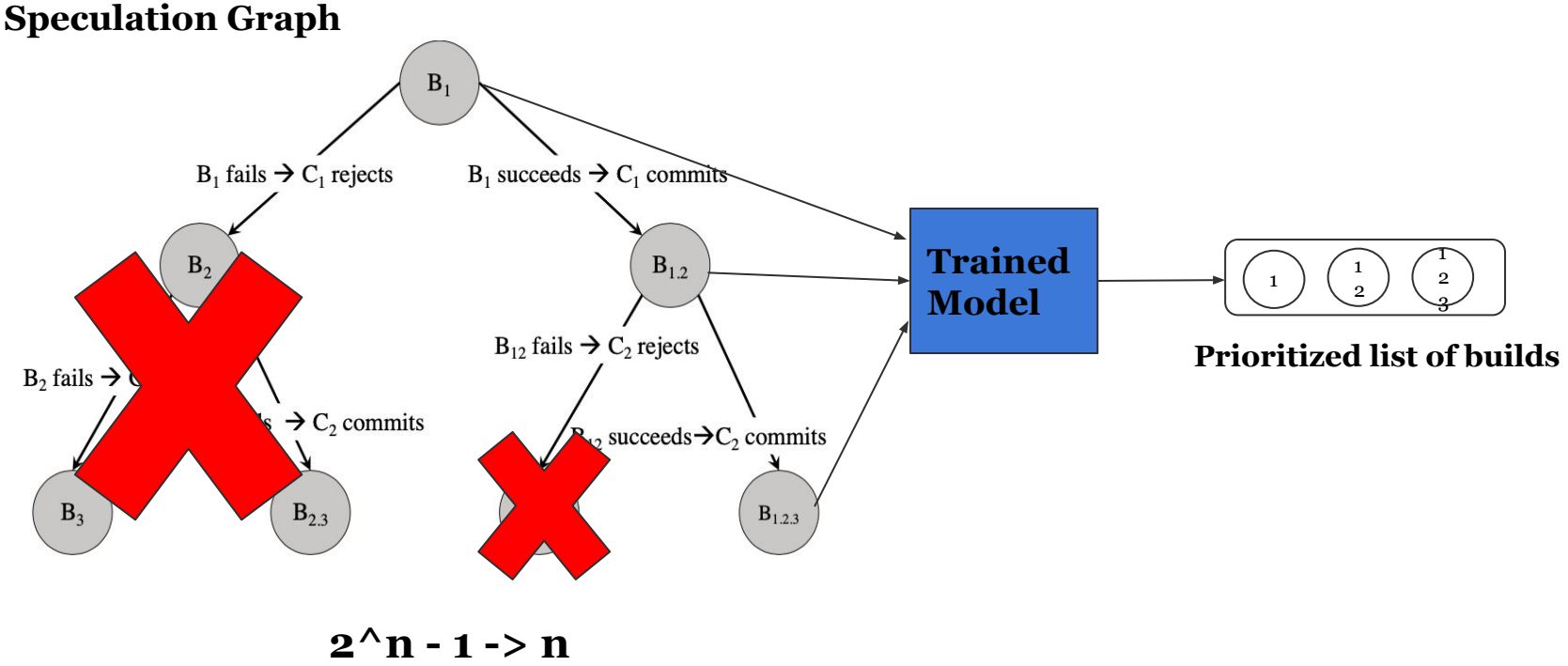
Overview of SubmitQueue Design

Idea behind:

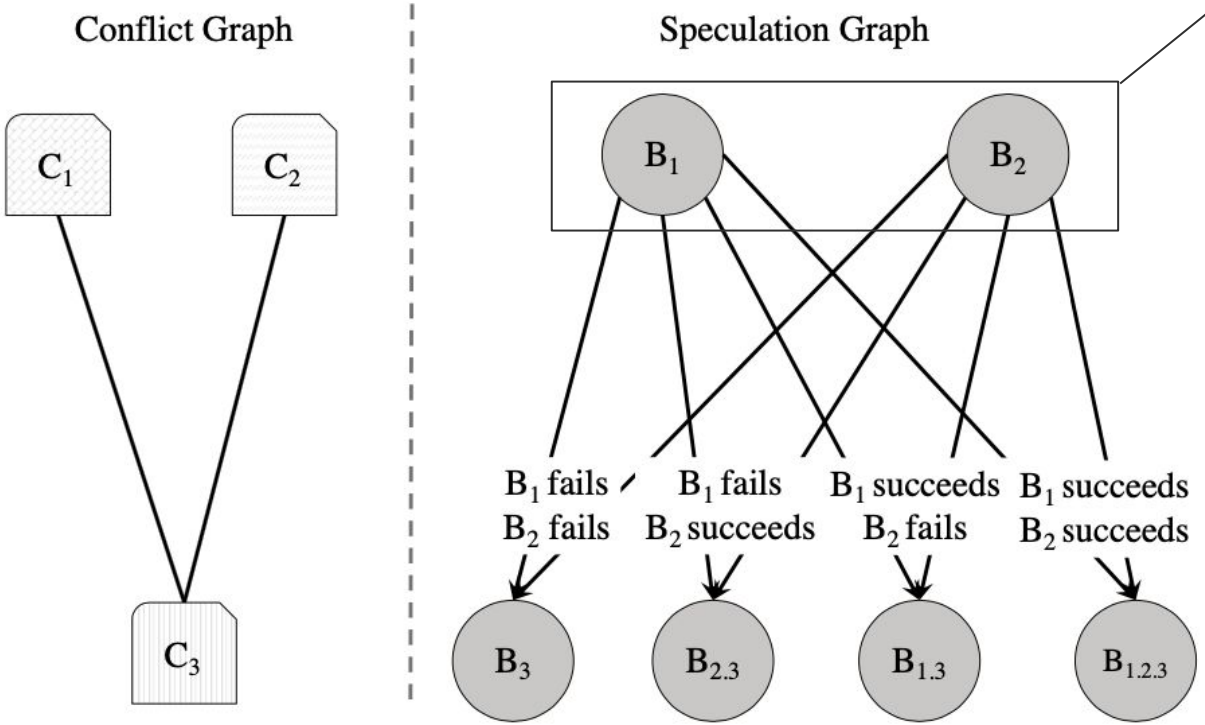
- Speculatively build possible outcomes for pending changes.
- Try to minimize the number of builds and parallelly build independent changes.



Speculation Engine



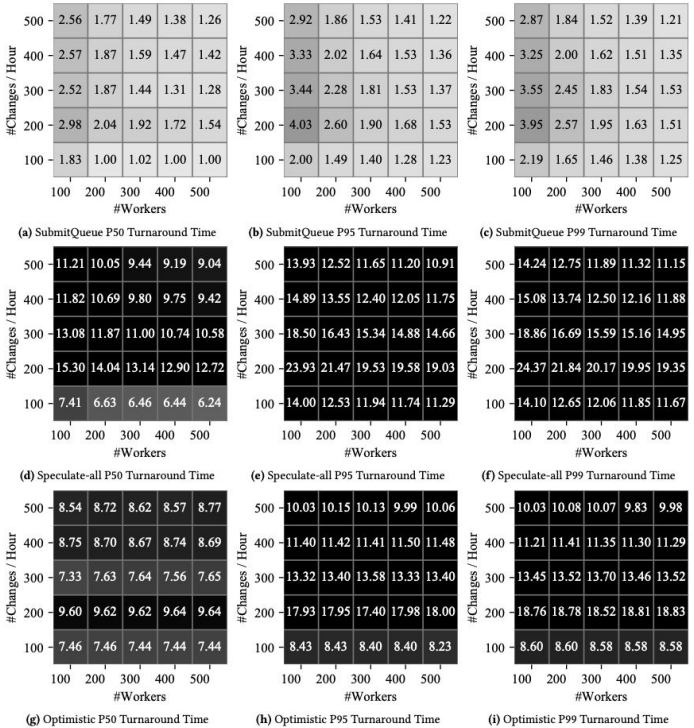
Conflict Analyzer

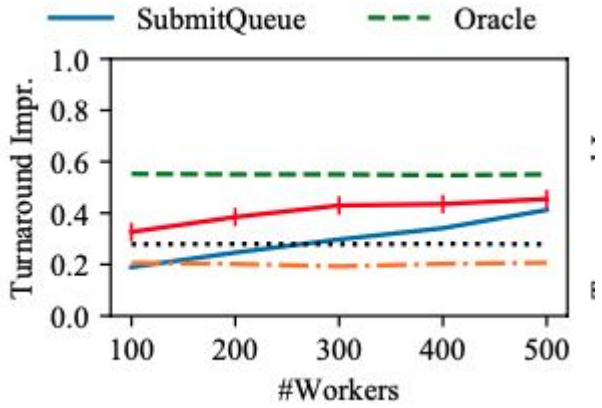


B1 and B2 are independent. No need to speculate B1,2 as B1,2 has lower probability to succeed than B1 and B2 individually

Observations

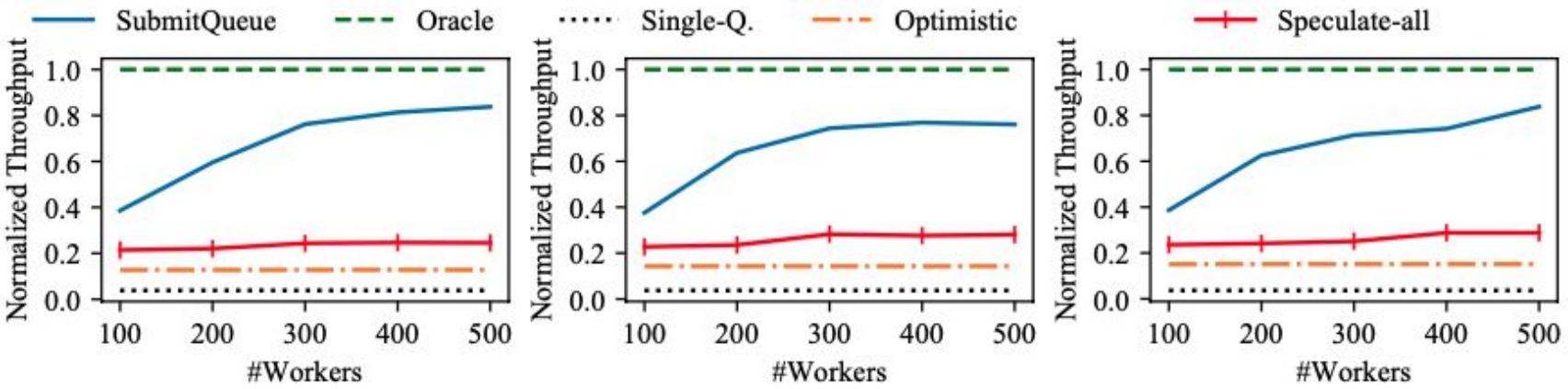
Turnaround Time:
Compare with Oracle who has theoretical shortest turnaround time, lowest overhead, and the best throughput.





Observations

Throughput



(a) 300 Changes / Hour

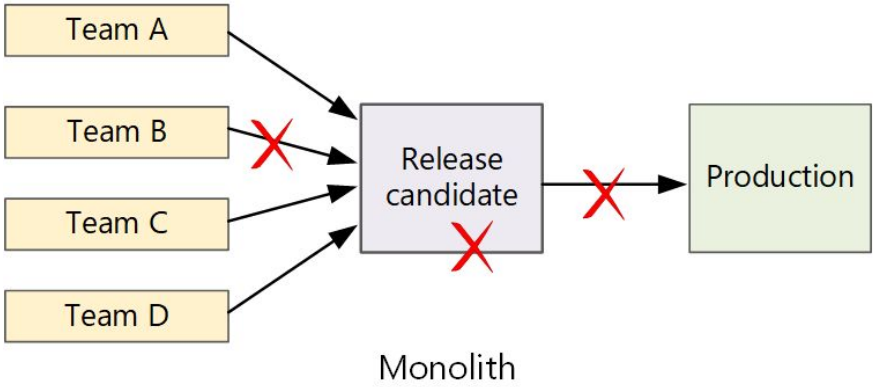
(b) 400 Changes / Hour

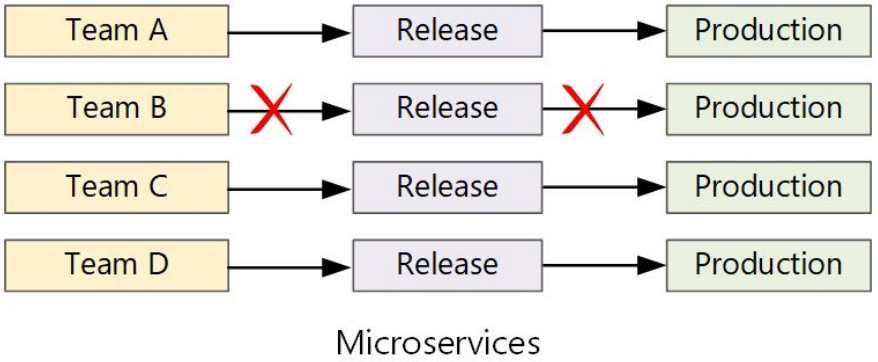
(c) 500 Changes / Hour

Limitation and Future Work

- No order for non-independent changes: Possible starvation for small changes
- Possible to implement a better machine learning method to train the prediction model
- Should abort a build which is near its completion but likelihood of success drops?
- Batch independent changes

CI/CD in Monolithic and Microservice System





- All development work feeds into a single CI/CD pipeline
- Bug in one team can delay the release of new feature
- Potentially high merge conflicts in CI and will get worse if system scale up
- CD is less complex but limited by the poor scalability of CI

- CI/CD pipeline in each team can build and deploy the services that it owns independently
- Team A's build failure does not impede Team B's release
- Less complex CI because of fewer merge conflict
- Hard to run end-to-end test in CD testing process
- May need a centralized release manager

Q/A

- What is SLA?
- How the speculation engine 97% of accuracy on predicting the outcome of builds with decision tree model?
- Mentioned in section 6, what artifacts does the cache store?
- Can build tasks get split into smaller pieces so that we may horizontally scale?