Distributed Grading Or: How I learned to Stop Worrying and Love The Scores My Peers Gave Me

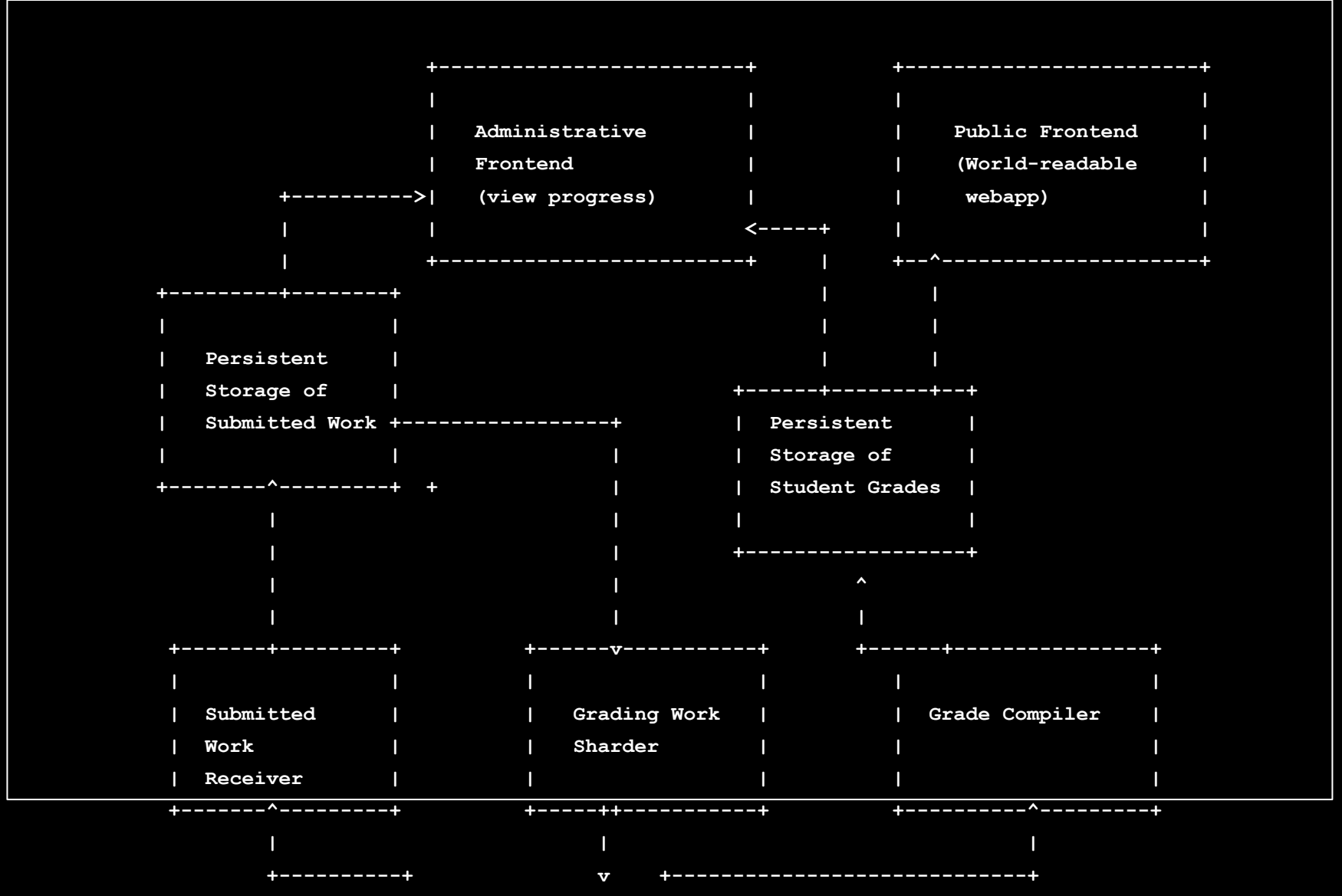
Jim Danz, Kenny Yu
Tony Ho, Stefan Muller
Willie Yao, Leora Pearson
Rob Bowden

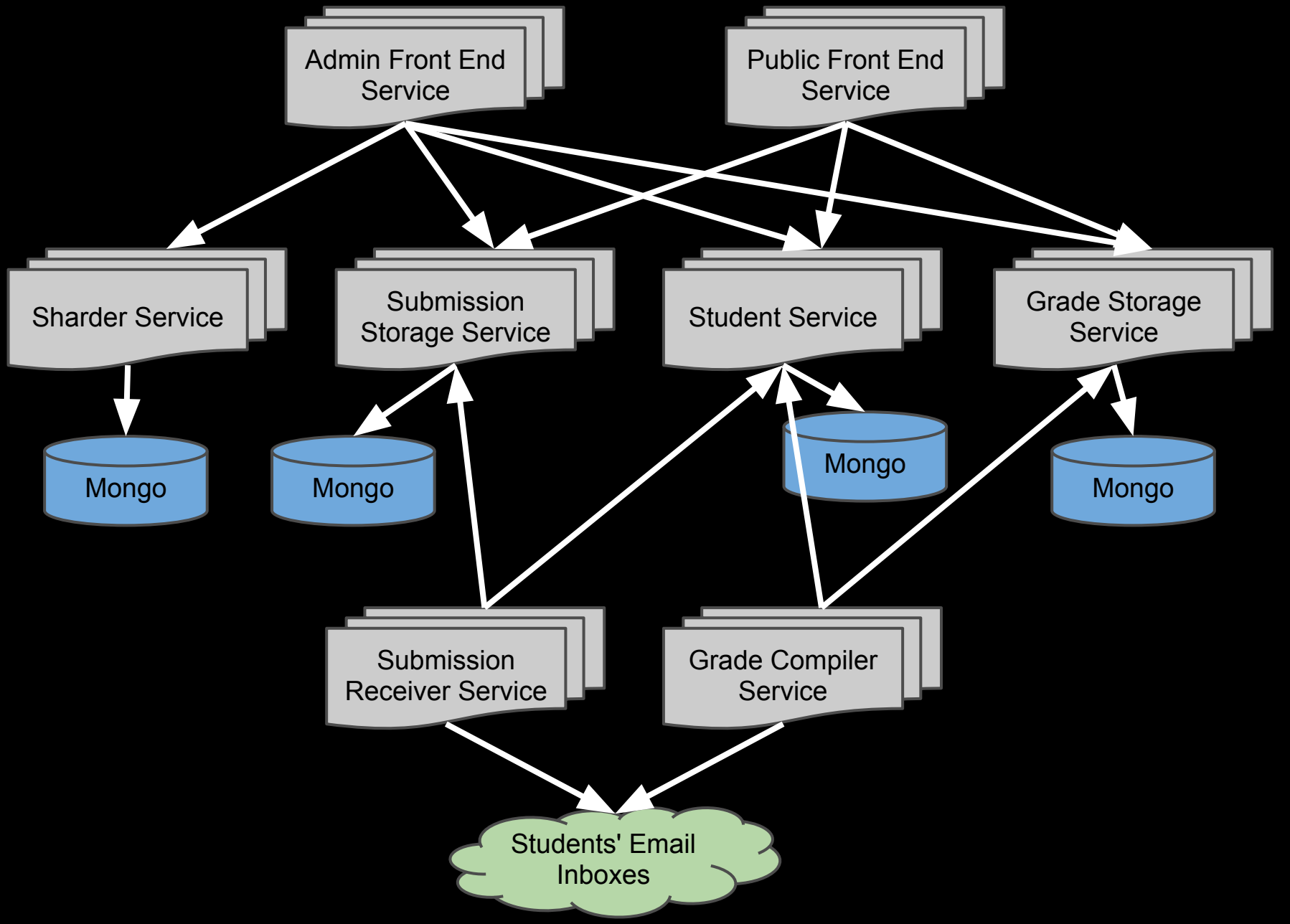
Problem Statement

- Students need to be assessed on their work.
  - Accountability
  - Ranking
- Teaching staff has limited resources
- Teaching staff has little personal incentive to be excited about grading student work
  - "I have to grade this stack of 20 papers, but what will *I* get out of it?"

Features Required

- Students submit their work
- Submissions are sharded among graders
- Graders submit their grades
- Admin interface, ability to manage students and assignments
- Ability to try out different "economic" models around the weight assigned to each grade


Administrative Frontend (view progress)
Public Frontend (World-readable webapp)
Persistent Storage of Submitted Work
Persistent Storage of Student Grades
Submitted Work Receiver
Grading Work Sharder
Grade Compiler
Students' Email Inboxes

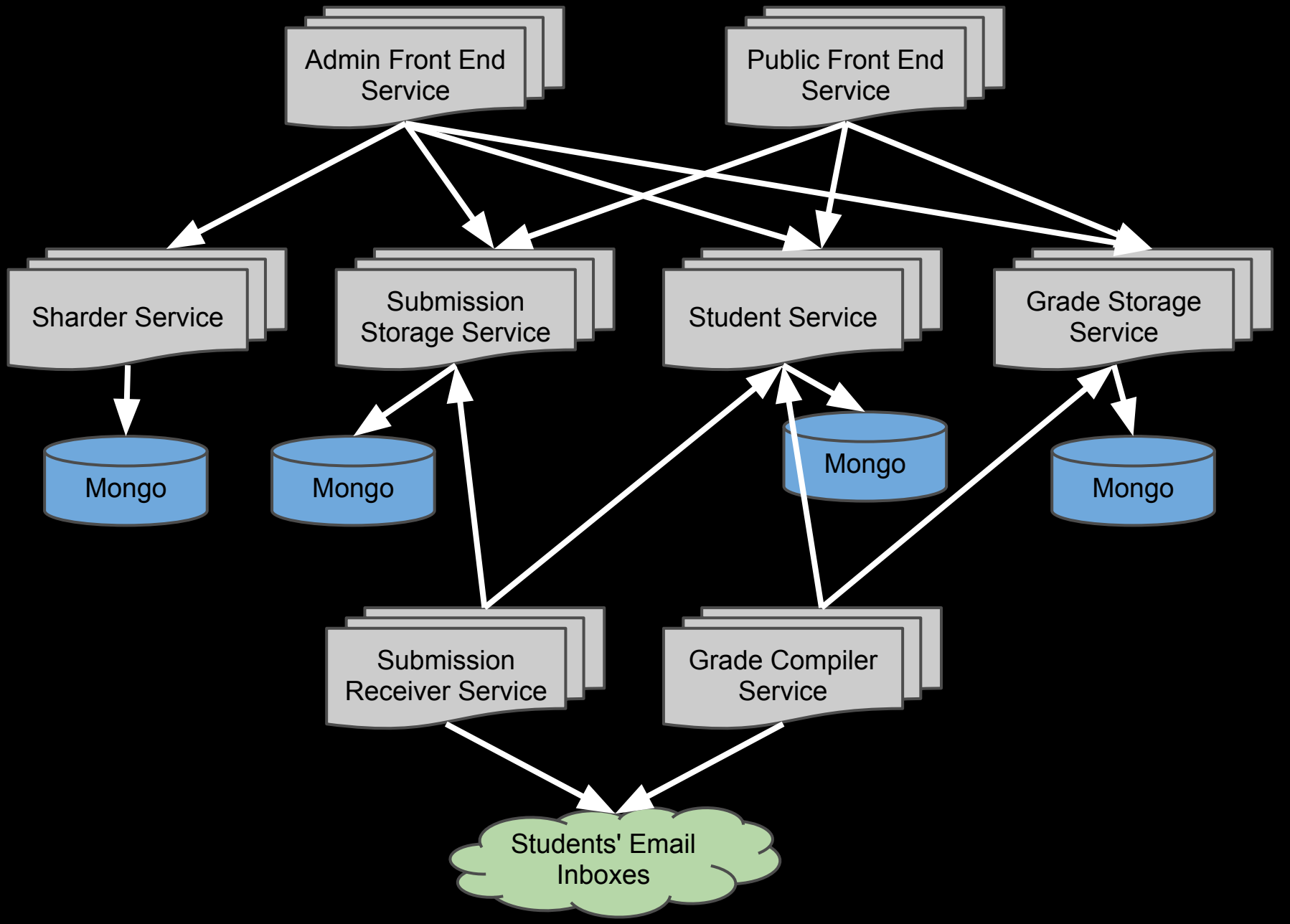

Admin Front End Service
Public Front End Service
Sharder Service
Submission Storage Service
Student Service
Grade Storage Service
Mongo
Mongo
Mongo
Mongo
Submission Receiver Service
Grade Compiler Service
Students' Email Inboxes

Assumptions about our system

- Synchronous communication (allows us to use timeouts)
- System can only tolerate crash failures
- Certain correctness relaxations are tolerable
  - eg, our "get latest submission for this (student, assignment) pair" method might return a slightly incorrect answer, if a student submits the same assignment twice with no delay inbetween, and the submissions get loadbalanced to different servers with out of sync clocks. But... if you submit twice within a second or two, you almost certainly submitted the same thing twice.

Design Considerations

- Service-Oriented Architecture
- Minimize the number of stateful services
- Distributed data storage, distributed at the conceptual / application level, rather than simply "let's just use a database that is distributed"
  - Four orthogonal datastores:
    - Submission Storage
    - Shard State Sorage
    - Grade Storage
    - Static Student Information


Admin Front End Service
Public Front End Service
Sharder Service
Submission Storage Service
Student Service
Grade Storage Service
Mongo
Mongo
Mongo
Mongo
Submission Receiver Service
Grade Compiler Service
Students' Email Inboxes

Technology Stack

- MongoDB for persistent storage
  - "hipster" distributed key-value store
  - Availability over consistency
  - Document storage
  - Fault tolerant
- Java for all application logic
  - RMI for communication between processes
- Http server for front end services

Distributed Systems Problems: Discovery

Discovery Mechanism

- configuration files
- contains a list of currently running (or possibly crashed) hostnames and ports for every service
- If service A wishes to communicate with service B, A finds *any* instance of service B and communicates directly with it.

Distributed Systems Problems: Fault Tolerance

Fault Tolerance of our Services

- We used stateless services
- Allows us to create many replicas of the same service without coordinating between replicas
- If service A wishes to communicate with service B, and the instance of B that A tries communicating with has crashed, A will choose another instance of B from the config file.

Distributed Systems Problems: Consistency

- Goal: avoid generating UUIDs at all costs
- Strategy: Hack the data model
- Define "uniqueness" based on a *(studentID, assignmentID, timestamp)* tuple
  - Again, slight differences in timestamp are irrelevant for a fixed studentID, assignmentID
  - We decide not to care about the "space leak" that gets created if a student submits many times (in a real system, we would cross that bridge if we got to it)

Distributed Systems Problems: Testing

Testing Strategy: 2 phases

- Phase 1: Local Testing
  - We create local instances of all the services, run our unit test suite to a mock database.
- Phase 2: Staging Environment
  - We create a real distributed system environment and run our integration test suite
  - Clean up our testing database afterwards
  - Difficult to automate this testing

Development Progress

- Implemented orthogonal persistent storage services using MongoDB
- RMI communication between services successfully implemented
- Fault tolerance made trivial using stateless services
- Front End - successful communication with back end services
  - still need to implement authentication
- We have unit tests and one-off integration tests, but we need to automate this process