CS 130 - Software Engineering

Spring 2025 – Instructors Michael Burns, Eric Hennigan, Alex Monroe, Philo Juang

Lecture 1: Development Environments and Source Control

Software Development builds big programs

- Large frameworks
- Developed by a team
- Documentation
- Entails:
    - Monitoring
    - Bug tracking
    - Unit, integration, and regression tests
    - Performance testing

Valuable software has risk

- Managing Risk
    - Revision control
    - Code reviews
    - Refactoring

Corporate Development Environments

- Identity
- Platform
- IDE
- Compiler
- File/Directory Structure
- Code Style
- Source Control System

Source Control

- Why?
    - Reproducible state
    - Documenting progress
- Tradeoffs
    - Adds complexity
    - Reduces Risk
    - More Scalable
- Changelist: Files in control organized by logic of project and build system
    - Self-contained
    - Small
    - Single function
    - Larger sequences arranged as branches
- Diffs: Instead of storing a copy of repo, store a diff that explains how to modify each version
- Commit: Snapshot of repository at a given point in time
    - Git collapses changes into a diff and gives commit a name
- Branch: Pointer to a specific commit
    - git checkout creates a new pointer that points to the same place currently pointed
- HEAD: Commit where **I** am currenting working
- Merge: Create a commit with two parents
    - Both parents' diffs are equivalent
- Rebase: Copy diff on top of the commit you are rebasing onto
    - Leaves an orphan commit
- Git pull –rebase: Pull new changes from remove repository

Lecture 2: Testing

Testing

- Unit Tests: For code and basic uses cases and errors
    - Isolated
    - Written in same language as the code they test
    - Fast to write and fast to execute
    - Serves as documentation
    - Easier refactoring
- Integration Tests: Used for binaries
- Regression Tests

Test-Driven Development: Writing tests first

- API prototyping
- When you know the desired result before you know how to implement it

Gauging how many tests to write

- Lots of people depend on it
- Failure is expensive or results in death
- More tests is not always the answer
    - Prototypes and rapidly changing code
    - Sometimes there's no substitute for manual testing
    - Tests are expensive to write and maintain

Config Parsing

- Separating binary and config increases robustness
    - Lets you roll back misconfigs easily
    - Introduces a layer in the code itself that needs to be tested

What should you test?

- Public APIs (one test case for each statement of the doc)
- Error handling

- Anything particularly complex
- Boundary conditions
    - Int Add(int a, int b) → What is a, b are 0, 1, negative, really big
- Pre and Post conditions: What is true before and after a loop, function, or conditional?
- Defensive: “What happens if something that ‘can’t happen’ happens?”
- Error conditions

Hard things to test:

- Global variables
- Objects with state
- Long, complex methods
- Tight coupling: Interdependent classes
- Bad abstractions

Lecture 3: Code Reviews and Web Servers

Source Code is not valuable unless someone else can use it

Code Review: Ensures someone else understands the code

Rubber Duck Debugging:

- Go through a problem step by step
- Think about each step clearly
- Realize what the problem is

Change Descriptions

- What is the change?
- Why was the change made?
- How was the why accomplished?
- Any new testing?

Code Review

- Flag errors of execution
    - Unclear documentation
    - Typos
    - Style violations
    - Bugs
- Apply thinking to find errors
    - Is algorithm correct?
    - Is this built to specifications?
    - Does this code need to exist?
    - Is this the most elegant solution?
- Develop shared understanding about the purpose of the code
    - Thanos Number: Should be maximized, number of people who understand code for a feature

Lecture 4: Build Systems and Deployment

Build: Compile code (Source code → Executable binary)

Deploy: Get code running (Run an executable)

Lessons

- Builds should be one easy step
- Builds should be repeatable
    - Different engineers at different times should get the same output
- Valuable things go in revision control

System Wishlists

- Correction: Builds should be a faithful representation of the current state of the source code
- Hermetic: Different users with different environments should get the same result
- Flexibility: Supports different languages, build rules, and compilation/linking options
- Ease of Use: Builds in one step
- Automatable: Builds should happen automatically and frequently
- Fast: If more builds are needed, they should be faster
    - Cache immediates, build cluster
- Integration: Build and test when you send code reviews
- Scalable: Supports large codebases with many users

Split Repository Strategy: Constrains the change rate of your dependencies

- Library built and released
- Release branches
- Builds are fast as you only build your own code
- Spend a week fixing things after release

Mono-repo Approach (Google)

- Development on HEAD
- Developers have the latest version

- Automated Tests
- Things are fixed as soon as they are broken
- You need to build the entire codebase to test your code
- Each commit changes a small part of the source tree

Containerization: A restricted view of the underlying OS

- See group of related processes, limit resource usage, restrict system calls

Docker: Way to build filesystems in layers

- Image: Snapshot of the OS, defined by commands
    - Created by docker build
- Container: Running instance of an image
    - Docker run
- Volume: Virtual Disk Image, persists across restarts and is sharable between containers
    - Docker run -v/--mount
    - Docker volume
- Bind Mount: Mount host file system in container, share data between host and container
    - docker run -v/--mount
- Port: Expose ports inside container to host and use Internet
    - Defined in Dockerfile/image (for documentation) or Container (for implementation)

Lecture 5: Testing, Refactoring, Dependency Injection

Testing

- Should be repeatable
- Refactor to allow for more opportunities to test
    - Handlers

Refactoring: Rewriting code to improve some property

- Can also focus on more maintainability or testability
- Ideally, create tests ahead of time so tests can verify correct behavior after changes
- Examples:
    - Extract methods when functions become too long or has boundary
    - Introduce parameter objects when a method's parameter list gets too long
        - Allows for creation of default params
    - Replace magic numbers with symbolic constants
    - Replace constructors with factory methods
        - Used when constructors are able to fail without exceptions
        - Useful for hiding which derived type is being returned
        - Forces you allocate object to heap
    - Introduce expression builders when object has required call sequence (stackable method calls to initially compile)

Dependency Injection

- Favor composition over inheritance
    - Storing objects in other objects rather than extending
- New keyword is bad
    - Creates dependencies specific to implementation
- Dependency Injection: Objects ask for what they need instead of retrieving what they need
    - Implementation agnostic (Objects depend on interfaces)

Lecture 6: DI Frameworks, Testing State, Debugging, Integration Tests

Dependency Injection Frameworks

- Injectors: Registers bindings to inject an instance of an interface
    - Ask injector for instances of objects instead of creating them yourself
    - Expensive to implement
- Use frameworks!

Mocks vs Fakes

- Real object: Same implementation you'd use in production
    - Preferred for testing as mocks and fakes can have buds and introduce more maintenance costs
- Fake object: Trivial implementation of the interface that satisfies all contracts but relies on memory only
    - Only need to be defined once (by service owner)
    - Fake, can be tested independently to define its behavior
    - Introduces maintenance cost
    - Not trivial to satisfy the service's contract in memory
- Mock object: Placeholder test object that can be set up to respond to specific stimuli and report back how it was interacted with
    - Only need to define the behavior you care about in your tests
        - Can confirm some API contracts
    - Introduces maintenance costs
    - Easy to test interaction instead of testing state
- It is highly recommended to test public APIs
    - A bad code smell is testing package private methods visible for testing
        - Generally indicates you're testing on some sort of interaction

Testing Readability

- DRY: Don't repeat yourself
- DAMP: Descriptive and Meaningful Phrases

Integration tests: Test code end to end

- Pick small handful of CRITICAL use cases for application

Debugging

- Access Logging: Web servers usually have access log that records:
    - Date, time, requestor IP, type of request, path, and response code
- Verbose logging: Triggered by certain conditions
- Logging Libraries: Boost → Trace trees
- Status Pages: Render log or present other statistics about the current process
- Operation Tracking: Collect operations by type (those which invoke the same handler)
- Distributed Tracing: Track single logical operation that crosses several servers in a distributed system

Continuous Build: Automate things you do yourself

Lecture 7: Static and Runtime Analysis

Static Analysis: Inspect code of program without directly executing it

- Compiler Warnings
- Type checking in the compiler: Gives compile time errors about illegal operations
- Linters: Helps catch nonportable constructs (typos/semantics)
- Abstract Interpretation: Model effect of statements on an abstract machine to identify mistakes
    - Walk the control flow graph, determine if invariants are broken
    - Catches use after free, uninitialized vars, deadlock
- Data Flow Analysis: Attempt to determine possible input values based on the control flow graph
    - Keep track of the set of possible values a variable can take at a given point in the program
    - Catches buffer overrun, dead/unreachable code
- Does not work if there are exponential number of paths through a program
- Extending the language rather than upgrading the language

Runtime Analysis: Run the program in an instrumented runtime

- Just a virtual machine that is checking for bad states in real time
- Slow, but hardware acceleration is possible

Lecture 8: Logging

Severity

- Trace/verbose/debug
- Info
- Warning
- Error
- Fatal → Crashes the server, killing all intermediate requests

Logging to disk

- Rotate log files
- Delete the oldest log if 10 or more exist
- Cap usage to 100MB

Privacy

- Aggregation: Derive counts to remove personally identifiable data
- Psuedononymization: Hah PII and store hash to count uniques

Error handling and recording information

- Crashing: Only when you cannot recover
- Boolean success return value
- Error Codes
- Exceptions
- Thread-local errno variable
- StatusOr: Return a class that contains an error status

Lecture 9: API Design

Web Server Architecture

- Request Handler: Encapsulates certain type of request behavior
- Request Container & Parser: Parses and represents requests
- Response Container and Generator: Incrementally builds up details of a response
- Mime Type Resolution: Generates specialized tasks

APIs (Application Program Interface)

- De-facto boundary between parts of software being built
- Extracts some generalizable functionality (3+) that is reused
- APIs are hard to kill, design errors are hard to repair
    - When in doubt, leave it out

Properties of Good API Design

- Ease-of-use
- Singular coherent concept
- Easy to extend/augment when needed
    - Implementation does not leak through interface

Properties of Bad API Design

- Does too many things
    - Kitchen sink: Gives clients even MORE power
- Usage is awkward
    - Annoying error handling
    - Requires careful orchestration
    - Small changes create big changes in behavior
- Makes things easier for users at the expense of clarity

Design Patterns

- Observer Pattern
    - Purpose: One-to-many notifications.

    - Good for: Instrumentation, policy layers.
    - Con: Can result in repetitive code.
- Lazy Initialization
    - Purpose: Delay heavy setup until needed.
    - Good for: Faster startup, simple init logic.
    - Con: Runtime unpredictability.
- Factory Pattern
    - Purpose: Centralized object creation.
    - Good for: Self-documenting constructors, dependency injection.
    - Con: Often overused.
- Singleton Pattern
    - Purpose: Ensure one instance.
    - Good for: Global state, system-wide access.
    - Con: Hinders testing, basically a global variable
- Object Pool / Free List
    - Purpose: Manage scarce or expensive resources.
    - Good for: Reusing e.g. DB connections.
    - Con: Complexity in ownership and thread safety.
- RAII (Resource Acquisition Is Initialization)
    - Purpose: Automatic resource management.
    - Good for: Memory safety and exception-safe cleanup.
    - Con: Verbose in GC languages.
- Decorator Pattern
    - Purpose: Runtime composition of behaviors.
    - Good for: Layered enhancements like compression or buffering.
    - Con: Layering can’t always be hidden.
- Continuation Pattern
    - Purpose: Callback-based flow.
    - Good for: Async workflows, composability.
    - Con: Can lead to “callback hell” (spaghetti code).
- Strategy Pattern

    - Purpose: Swap algorithms or behaviors.
    - Good for: Flexible polymorphism.
    - Con: Too many classes, less transparent control flow.

Antipatterns:

- Stringly-Typed APIs
    - Issue: Using plain strings instead of meaningful types.
    - Consequence: Hard to read, maintain, and validate.
- Unclear Object Lifetimes
    - Issue: Manual memory management without ownership semantics.
    - Fix: Use smart pointers like unique_ptr

Lecture 11: The Art of Readable Code

Fundamental Theorem of Code Readability

- Code should be written to minimize the time it would take fore someone else to understand it

Aspects of Coding

- Code Length
    - Fewer lines is code is generally better
    - "Function should be at most 1 screen of code" is arbitrary
- Naming
    - Put units in quantity names
    - Add prefixes or suffixes to clarity important attributes
    - Use type systems
    - Names should be searchable, distinct, pronounceable, and meaningful
    - Give complicated comparisons names
- Commenting
    - Be concise, sometimes code speaks for itself
    - Be robust/specific
    - Do not comment built-in language features, restate function names, or restate algorithms
- Comparison Expressions
    - Left hand side: Expression being interrogated, value is more in flux
    - Right hand side: Being compared against, value is more constant
- Unnecessary Variables
- Avoid Complexity
    - Code should be organized to do one task at a time
    - Solve the opposite problem
- Avoid Nesting
    - Guard with Early Returns

Lecture 12: Threading and Concurrency

Refreshers

- Threads have shared memory
    - Corrupting memory from one thread will affect all other threads
- Processes are functionally independent
- Synchronization primitives
    - Mutex: Protects resources in first come first serve manner by requiring holding onto a certain object
    - Semaphore: Restricted parallelism using a capacity
        - Ex: Access to a database (thousands of connections)
- Critical Sections: Piece of code, where if you run multiple threads, there can be issues
- Race Conditions: Reliance on processes completing in certain order
    - Two processes running at same time. You have no idea if one completes before the other.
- Deadlock: Thread A waits for Thread B. Thread B waits for Thread A.
- Concurrency: Makes things faster by keeping things busy

Concurrency in the Real World

Motivating Case: CS 130 website

- Incorrect writes if two users push changes at the same time (Server is a shared resource)
- Solution: Use a unique identifier instead of a shared file directory
    - Better as we remove the concept of a shared resource
- Solution: Lock and block until other user runs

Dealing with Concurrency

- Strategy 1: Be slow. Conclude we don't need to be concurrent
    - Lesson: Always profile before you optimize
        - If you are going to incur the cognitive overhead of using concurrency, you should ensure its worth it

    - Example: Regression Test for ML models
        - Copy models to a staging directory
            - Copy baseline models
            - Copy test models
        - Evaluate the models on the same inputs. Compare the outputs
        - Doesn't make sense to use concurrency, as it'll take the same amount of time with or without it.
- Strategy 2: Isolate
    - Idea: If there's no shared resources, there is no difference between single-noded architectures. This eliminates the cost of concurrency
    - Lesson: Push concurrent code into "no-shared resources" blocks
    - Example: MacOS
        - Initially required developers to yield to the OS
            - Terrible API: Complex for developers and crash-prevalent for users
        - Running code would block the cursor for moving for instance
    - Example: Web Indexer
        - Input: Crawled web pages
            - Indexer analyzes each web page, writes what it learned to a database
        - Process each document in separate threads
            - Concurrency code is isolated to main loop
            - Rest of code is single-threaded
            - Emphasizes throughput more than latency
    - Example: Web Server
        - Run each request in a separate thread
- Strategy 3: Use a library
    - Example: Threads in C++
        - No parameters on functions
        - Cout has no blocking and simply dumps characters into a buffer
            - Causes race conditions
        - Pass by const reference or ptr

                - *That* function can't change data, but another thread can
                - Promotes serializability
            - Pass a non const pointer
                - "It's toast"
    - Example: RPC Server
        - We've already isolated each request in its own thread
        - But we want to make things faster
- Strategy 4: Use a mutex
    - Perform an operation and modify it. Write it back to entry
        - Read and write should not interleave
    - Constructors are not atomic
- Strategy 5: Never forget to unlock
    - Lock_guards (does not lock the mutex directly)
- Strategy 6: Use tools
    - Clang thread safety analysis
        - Add annotations which are checked at compile time
- Strategy 7: Comment your code
    - Add comments about thread safety

Lecture 13: Web Servers and Distributed System Architecture

Supercomputers

- 2000 Netra: Redundant Hardware, built-in monitoring, Hot swappable
    - Tolerates and detects hardware failure
    - Physically constrained
- Today: Distributed Systems
    - Networks are faster
- Datacenter Efficiency:
    - Power use is up, increased workload is more efficiently executed

Distributed Applications

- Reverse proxy: Type of proxy server that retrieves resources on behalf of one or more servers
    - Resources appear as if they originated from proxy server itself
    - Intermediary for its associated servers to be contacted by any client
- Forward proxy: Intermediary for its associated clients to contact any server, acts on behalf of client
- Maglev: Fast, Reliable Load Balancer
    - Extremely fast hashing and memory
    - Total control of Network interface controllers
    - High resistance to CPU cache misses
    - Load shedding
- GFE: User connects to
- BGP takes the most specific broadcast address
    - Youtube and Pakistan Telecom

Distributed Application Examples

- MapReduce: Data Processing on Large Clusters
- BigTable: Distributed Storage System for Structured Data

- Spanner: Use TrueTime to create causal ordering

Distributed System Design Patterns

- Manager/Worker: Central coordinator (manager) dispatches to workers.
    - Useful for parallel data
    - Divides load across several identical jobs
    - Load balancing, MapReduce master, BigTable master
- Dynamic Sharding: Load-based splitting of work.
    - Useful for when single worker maxes out
        - Divide worker's problem across several other workers
    - Easily add more works to increase assigned resources
    - Reallocation of map shards
- Separate Routing/Data Path: Manager routes, backend serves.
    - Route maintained by node that client queries
    - Clients directly communicate with backend (that has data)
    - Increases total ingress/egress bandwidth
    - Load Balancers
- Admission Control: Drop excess requests to avoid overload.
    - Make sure request can be serviced before accepting request
        - Prevents memory exhaustion from queueing requests
    - Bigtable Pushback, Load Balancer
- Distributed Consensus: Paxos/RAFT used for agreement.
    - Allow group of distributed processes to agree on state change
        - Helps with network partitions with unreachable nodes
    - Spanner transactions, Bigtable master election
- Queuing: Decouples producer and consumer rates.
    - Allows for systems to process work at different rates
    - When consumers are slow, work is queued durably
        - If consumers fall behind, so do producers
    - BigTable Commit log
- Memcache: Temporal caching to save compute.

    - Avoid recomputing same result many times within some time window
        - Works well with temporal locality of requests/small working set
        - Can save CPU or Spindles
    - BigTable block cache
- MVCC: Supports concurrent reads/writes via timestamped versions.
    - Multiversion Concurrency Control
    - For recent writes, store timestamped versions
        - Allows interleaves reads and writes without exclusive locks
        - Requires garbage collection to clean up duplicates
    - Percolator Distributed Transactions, Spanner Snapshot Reads

Distributed System Anti-Patterns

- Storing metadata within system
    - When system is loaded heavily, metadata may not be accessible
    - BigTable Metadata
- Global config rollout: One bad config can crash all processes.

Known Hard Problems in Distributed Systems

- CAP Theorem: Can't simultaneously have Consistency, Availability, Partition tolerance.
- Atomicity Domains: Difficulty in coordinating operations across services.
    - Make downstream operations idemptent (replayable)
- QoS vs. Utilization: Tradeoff between resource control and efficiency.
- Rollouts: Safe protocol upgrades need backward compatibility.
- Reprocessing: Make data appear like it uses latest code
    - Too much data to recompute in a reasonable amount of time
        - Compute some part of dataset
- Data Moves: Physically relocating systems is often impractical.

Common Failure Modes

- Correctness Bugs: Easier to catch with testing.
- Performance Bugs: Require full-scale testing.

- Rate Mismatch: Can cause nodes to crash despite admission control.
- Hotspots: Small working sets can overwhelm specific nodes.
- Death Rays: Overloads that follow nodes after recovery.
- Amplified Rare Events: At scale, rare bugs become frequen

Lecture 14: Performance

Determine exit criteria from when a component is fast enough

RAIL (Each can be scored)

- Response: Any kind, can be minimal
- Animation: Time for animations to begin
- Idle: Time for page to reach a steady-state
- Load: Time it takes for all data to be on the user's device and ready to run

Performance Monitoring:

- Agree upon some semi-stable testing environments
- Gather inputs that affect performance
- On input changes, run app N times while measuring stuff
- Archive and track results over time
    - Know quickly when app gets slow (and why)

With servers, you can throw money at the problem

Issues with reproducibility:

- Users have different hardware, OS, user-specific data, and network bandwidth
- Profile users in a realistic way (Google UMA)

Pretend things are fast

- Loading shims
- Progressive enhancement
- Progress bars

Optimization

- Lazy execution
- Reduce conditional rendering
    - Total Render time = Number of items rendered * Cost of rendering each thing
    - Reduce the cost of rendering

        - Reuse list elements
        - SVGs are expensive
        - Break down giant regexes
        - Cut the fattest part

Capacity planning

- If queue overflows, just spin up more jobs
    - Automation

Debugging Distributed Systems

- Status Pages
    - Each process serves over HTTP
    - Can expose state of internal data
        - Stacktrace of live threads
        - State of all currently processing RPCs
- Logging
    - Log all the time as we don't know when errors occur
    - When and what requests arrive
    - Details about erroneous conditions
    - What part of the code is involved
- Local Request Tracing
    - Maintain request logs (traces) for interleaved events
        - Kept for active requests (less useful for retrospective debugging)
- Distributed tracing
    - Token representing logical operation is passed along with each RPC
        - Stats are logged by each process and system aggregates data
    - Used to measure latency and resource usage

Lecture 15: Monitoring, Documentation, Postmortems

Monitoring

- Maslow's Hierarchy of Needs
    - Self actualization is the goal
- Site Reliability Engineering's Pyramid of Productionization
    - Successful product is the goal
    - Monitoring is the foundation

Production

- Job: Process running on a machine, instance of a service
    - Docker container
- Service: Collection of identical jobs
    - Docker image

Monitoring

- Monitoring process collects data from each job
    - Times and running data are collected for every job and service
    - Stored as epoch timestamps
- Black box monitoring
    - OS-level monitoring
    - Agnostic of application internal state
        - CPU usage
        - Disk space
- White box monitoring
    - Application level monitoring
    - Exposes application internal state
        - Parsing logs or URLs
        - Exceptions
- Client interface
    - Query language for retrieving time series and transforming them

Using Monitoring: Alerts

- Detection of failures: Server crash, failure of remote service, errors being served to users
- Alert Components
    - Condition
    - Time Period
    - Severity
    - Message

Service Level Objectives (SLO)

- What is our goal for the uptime of our service?
- Strict SLO
    - More people oncall
    - Engineers in different timezones
    - Consider long term work/life balance
    - More training and practice for people on call
- Loose SLO
    - Feature Development
    - Prototyping
    - Major architectural improvements
    - Crazy ideas
- Downtime
    - New features
    - Changing configuration
    - Releases
    - Growing user base
- SLO as a risk budget
    - Too little downtime has risk
        - Strain on on-call team
        - Too little time spent developing features
        - Client assume service is too reliable
        - Lack of error handling when service is down

Minimizing toil:

- Manual vs Repetitive
    - Doing something for the first time is a learning experience
    - Doing it more the 100^{th} time implies an automation is needed
- Tactical vs Strategic
    - Solves an immediate issue rather than the underlying problem
    - No enduring value: No long-term improvements to the system
    - O(n) with service growth: More work as your system grows

Documentation

- Product Requirement Documents:
    - Describes the problem and a product-level solution to the problem
    - Vision, Motivation, Goals, Requirements, Success Criterium
- Technical Design Documents:
    - Describes the implementation of the product-level solution
    - Objective, Background, Requirements, Detailed design, Alternatives considered
    - Extra Credit: Security, privacy, testing plan, work estimate
- API documents:
    - How to use the code, what are the classes, what do the methods do
- Good requirements are unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable for the lifetime of the projects
    - Avoid ambiguous language (could, may)
    - Use strong language (shall, will)
    - Quantify verification criteria

Lecture 16: Postmortems, Team Structureu

Postmortem Structure:

- Title
- Summary
- Impact
- Root Cause
- Timeline (including beginning and ending of outage)
- What went well/poorly
- Action items

Sometimes preventing errors is more expensive that having them

Ask others for postmortems

Retrospectives: No outage, similar to a postmortem

- What went well
- Fixes 1-2 things
- Way to deal with overzealous postmortem action items

Development Considerations

- Economic
- Environmental
- Social and Cultural
- Political and Legal
- Ethical
- Health and Safety
- Sustainability
- Usability

Team Structure

- Definitely on team

    - Engineers
    - Management: Ensures you are productive and creates an environment where you can be happy. Sometimes technical
    - Tech Lead: Not a manager, but in charge of the technical direction of project
- Likely to be on team
    - Product Managers: Understands the market and interface with engineers
    - Testers: Automated testing, manual testing, normally have "Engineer" in title
    - UI Designers: Builds the product in Photoshop, gives metrics of design properties
    - UX Researchers: Sits behind a one-way mirror and watches people use product''
- Sometimes on the team
    - Project Manager: Some technical, others not. Coordinates and checks up on how work is going
    - Architect: Tries to define a technical vision
    - Operations: Keeps services running. Sometimes carry pagers
    - Tech Writer
    - Release Engineer
    - Researchers: Mainly focused on long-term research
- Business
    - Executives/Upper Management: In more mature teams, these people are removed
    - Marketing: Sometimes called PMs
    - Sales
    - Product Experts: Dressed really nicely, makes sales to customer before product is developed
- Human Resources
    - Escalate where your coworkers are doing something wrong

Career and Longterm

- Fix bugs → Larger scale changes → Design features → Design systems → Think about services → Start new projects
- Development philosophies
    - Waterfall

        - Father requirements → High level design → low level design → Coding → Testing → Release
    - Agile
        - Short sprints on the order of weeks
        - Keep quality high at all times
- Avoiding burnout
    - No all nighters
    - Do not work on weekends
    - Unplug on vacation
        - Plan early, broadcast dates, do not respond
    - Work life balance
        - Scheduled Send
- Negotiation
    - Every interaction is a negotiation
    - Focus on your goals
    - The most important person in a negotiation is them
        - Really try to understand what the other side wants
    - Make emotional payments
        - Use empathy, apologies. People are not always rational
    - Be incremental
    - Trade things you value unequally
    - Find their standards/policies
    - Be transparent and constructive
        - “We both want this project to succeed”
    - Prepare

Lecture 17: Deployments, Experiments, Launches

Deployments

- Development Types
    - Devel(op)
        - Has latest version of code
        - Used by developers to test new functionality in a deployed setting
        - Usually connects to development backends
    - Staging
        - Features are deployed to stage before production
        - Useful for QA testing to gain confidence that deployment will work
        - Connects to production backends where possible
    - Canary
        - Handles small amount of user traffic
        - Outages/downtime are less costly
    - Prod(uction)
        - Hosts all user traffic
        - Outages or downtime often leads to huge costs

Rollouts

1. Stop directing traffic to some small set of servers (via a load balancer)
2. Restart those servers to a new version
3. Restore traffic to new servers
4. Repeat until all servers are processed

Version incompatibility

- Systems are often composed of multiple binaries
    - What code is running that calls me?
    - What code is running that I am calling?
    - Will everything still work if I have to roll back to an old version in a few days?
- Protocol Buffers: Define cross binary communication objects outside of any one binary
    - Generate different representations of data on the fly

Updating proto:

- Backwards Compatibility: Ensure older versions of the prototype still work with the new definition
- Avoid during Protocol Buffer:
    - Do not delete a field
    - Do not change the type of a vielf
    - Do not reuse IDs
- Thus,
    - Add fields
    - Mark fields as deprecated
- Modify a field
    - Add new field you want to use
    - Start populating both fields (new and old)
    - Switch code to read/write new field
    - Remove all usages of old field
    - Stop populating old field
    - (optional) Remove old field
- Forwards Compatibility: Ensure newer versions of prototype work with older binaries
    - Ignore unknown fields at deserialization time
    - Ensure code does not send new fields until all of dependencies are updated

Feature Flags

- Enable/disable features at runtime without redeploying.
- Useful for gradual rollout, dependency coordination, and quick disabling.
- Should be deployment-specific and independently updatable.

Experiments

- Setup:
    - Control Group: No change, serves as baseline.
    - Treatment Group: Receives new feature.
- Diversion Techniques:

    - Use cookies, sessions, user IDs.
    - Diversion can be at user, video, or request level.
- Counterfactuals:
    - Technique to measure impact of features affecting only a subset of users.
    - E.g., only measure users who clicked a certain button.