Engineering Core Principles

In all of our learning and development activities from onboarding, to structured training sessions, to brown bags, to direct mentoring, it helps to have a core set of principles to fall back on. You may teach someone about SQL migrations during onboarding not merely because "That's just what we do here", but because we have *Empathy* and because we take *Tiny Steps* to *Limit Risk*.

We use these bedrock principles when creating training material, when coaching teammates, and when making decisions throughout our own work. If a teaching topic can't be tied back to these principles we have to ask, "Should we be teaching this?". If a technical decision can't be tied back to these principles we have to ask, "Should we be making a different decision?".

We have *Empathy*, take *Tiny Steps*, *Limit Risk*, and *Seek Evidence*. We get things *Right Once*.

Empathy

We have empathy for our teammates, for the client, and for ourselves.

- We detect and respect our audience's communication style. We don't distract people with excessive technical detail. We choose language to make our case effectively and guide decision-makers to the right conclusions.
- We respect our peers' time and workload. We find ways to limit interruptions. We create a clear commit history and submit our work for peer review with informative pull request descriptions. We submit work that is easy to review.
- We recognize skill gaps and provide mentorship.
- We respect the client's mission. We avoid ivory tower coding perfection, always keeping the client's needs and goals in mind.
- We care about the impact we have on a client's own developers, and guide them towards understanding and success.

Tiny Steps

We approach our work in small, safe steps.

- We take a feature and break it down into smaller and smaller pieces until it's easy to estimate.
- We make frequent commits with meaningful commit messages. We are mindful of our own train of thought.
- We break large feature development into multiple, smaller pull requests, easing code reviews and merges, improving throughput, and communicating a clear sense of what is Done.

Limit Risk

We limit the number of times we have to be careful, because the alternative guarantees failure.

- We create repeatable build and deployment processes, promoting tested changes from development, to staging, to production.
- We have an informed and healthy fear of making live changes against production.
- We evaluate our decisions with a clear understanding of *"What could go wrong?"*
- We write meaningful automated tests throughout development, to ensure that not only does a feature work upon delivery, but that it keeps working in light of later changes.
- We review each other's work to catch problems early.

Seek Evidence

We abhor "guess and check" troubleshooting. We use the evidence in front of us, and take steps to uncover more evidence when working through a tough problem. We arrive at solutions faster by slowing down and being methodical in our problem-solving.

- We read and seek to understand error messages.
- We dig deeper into stack traces and the like for more information.
- We search for existing solutions. "Surely someone's run into this problem before."
- We create new evidence with informed experiments.
- When the sky is falling and the evidence doesn't make any sense, we use that fact to stop and check our assumptions. Even a "wrong" error message is telling us something.

Right Once, Fun Everywhere

We avoid redundant work. We recognize that painful development experiences from past projects and past jobs don't have to be the norm. We learn from those pains, taking action now to make our lives easier later. We solve problems in such a way that they *stay solved.*

- We automate painful, error-prone work.
- We set up build, test, and deployment processes at the start of a project.
- We set up informative logging because we know it will help with troubleshooting later.
- We *respect* the DRY principle ("Don't Repeat Yourself") to avoid wasteful coding and the risks that come with code duplication, but we recognize that blindly following the rule can be harmful, too. Rather than focusing merely on resolving duplication in code snippets, we identify and pull out *useful* abstractions that drive us toward our goals. We evaluate the risks of duplication through the lens of our client's mission.
- We recognize tried-and-true patterns and release them as open-source tools, easing the startup of new projects and enabling team members to get oriented quickly as they move from project to project.

- When we need to keep on remembering to get something right, we seek out a solution that makes getting it right the *default*.
- Instead of modifying a CREATE TABLE script for local development and writing an equivalent ALTER TABLE for deployments, we use SQL migrations. We write a single change script appropriate in all environments. We get it Right Once, resulting in smooth merges with our teammates' work and smooth deployments to all environments.
- Instead of writing a developer-facing build script and separately defining a TeamCity configuration with many build steps (compile, run tests, …) we write one build script appropriate in all environments. Continuous Integration setup focuses on one key step: "run the build script". We get automated builds Right Once, and leverage that single process in all environments.