AI Dev Team Playbook

Building High-Quality Software with Junior Developers in an AI-Augmented World

Prepared by Obed Industries
Version 1.0 — 2026

> ***Who this is for:*** *Startup founders and CTOs who are hiring junior developers and want to ship quality software fast — without chaos. This playbook covers how to set up your team's workflows, standards, and culture so that AI tools like GitHub Copilot accelerate your team without cutting corners on quality or safety.*

Table of Contents

Introduction: Why This Playbook Exists

Junior developers in 2026 are not the junior developers of 2015. Today, a 22-year-old fresh out of a bootcamp can write production-quality boilerplate in minutes using GitHub Copilot. That's the good news.

The bad news: speed without structure is how bugs reach production. AI tools amplify both the good and the bad habits of your team. If your team doesn't have clear standards, AI will help them write messy code faster. If your team *does* have clear standards, AI becomes a genuine force multiplier.

This playbook is your blueprint for the latter.

It assumes: - You're a small startup (2–15 developers) - You're using GitHub (or GitLab — the principles apply) - You have GitHub Copilot or a similar AI coding assistant - Some of your code affects safety-critical outcomes (medical, transportation, financial, education — anywhere a bug has real-world consequences)

Let's build something you can be proud of.

---

1. Git Workflow & Branching Strategy

The Core Principle: Protect `main`

Your `main` branch is sacred. It should always be deployable. Always. If someone is scared to push to main because "it might break something," that's a symptom of missing process, not missing courage.

Start here. On day one. Before you write a line of code.

```
Settings → Branches → Add branch protection rule for "main":
  ✓ Require a pull request before merging
  ✓ Require at least 1 approval
  ✓ Require status checks to pass before merging
  ✓ Require branches to be up to date before merging
  ✗ Allow force pushes (OFF)
  ✗ Allow deletions (OFF)
```

Trunk-Based Development vs. GitFlow: Choose One

For most startups: use Trunk-Based Development.

The choice isn't religious — it's practical. Here's the honest breakdown:

	Trunk-Based	GitFlow
Best for	Startups, fast teams, CI/CD pipelines	Enterprise, long release cycles
Branch lifespan	Hours to 2–3 days	Days to weeks
Merge conflicts	Rare	Frequent
Junior-friendly	Yes (simpler mental model)	Can be confusing
AI tooling fit	Excellent	Okay

Trunk-Based Development means everyone branches off `main`, works in small increments, and merges back quickly. Feature flags handle unfinished features in production. This is what Google, Facebook, and most high-velocity startups use.

GitFlow has long-lived `develop`, `release`, and `hotfix` branches. It's useful when you have strict versioned releases (like a mobile SDK or an enterprise product with quarterly releases). Most early-stage startups don't need it.

Recommended Branching Structure (Trunk-Based)

```
main
├── feature/TICKET-123-add-video-scoring
├── feature/TICKET-124-dashboard-filters
├── fix/TICKET-125-null-pointer-on-upload
└── chore/update-dependencies
```

Branch naming conventions:

```
feature/TICKET-ID-short-description
fix/TICKET-ID-short-description
chore/brief-description
hotfix/brief-description
docs/brief-description
```

Rules: - Branch names are lowercase, hyphenated - Always include the ticket/issue number when one exists - Keep descriptions short (3–5 words max) - Delete branches after merging (GitHub can auto-delete merged branches)

The Feature Flag Pattern for AI-Heavy Teams

When your AI tool (Copilot, etc.) helps a junior dev write a bigger feature than expected, it's tempting to keep the branch alive for days while it grows. Resist this. Use feature flags instead:

```
# config/features.py
FEATURE_FLAGS = {
    "new_scoring_algorithm": os.getenv("FF_NEW_SCORING", "false").lower() == "true",
    "enhanced_dashboard": os.getenv("FF_DASHBOARD", "false").lower() == "true",
}

# usage
if feature_flags.is_enabled("new_scoring_algorithm"):
    result = new_algorithm.score(video)
else:
    result = legacy_algorithm.score(video)
```

This lets you merge incomplete features to `main` safely, deploy without activating them, and turn them on per-environment. It also makes rollback trivial.

Commit Message Standards

Good commit messages are not optional. They're your team's changelog, your debugging breadcrumbs, and your code review history — all in one.

Use the **Conventional Commits** format:

```
<type>(<scope>): <short description>

[optional body]

[optional footer: TICKET-123, BREAKING CHANGE, etc.]
```

Types: - `feat`: New feature - `fix`: Bug fix - `refactor`: Code change that neither fixes a bug nor adds a feature - `test`: Adding or updating tests - `docs`: Documentation changes - `chore`: Build process, dependency updates, tooling - `perf`: Performance improvements

Good examples:

```
feat(scoring): add confidence threshold filter to video evaluation

fix(auth): handle null token on session expiry — closes TICKET-98

refactor(api): extract video upload logic into VideoService class

test(scoring): add edge case tests for zero-frame video input
```

Bad examples (do not accept these in review):

```
fix stuff
WIP
updates
asdf
Copilot suggestion
```

> ***AI Tip:*** *GitHub Copilot and ChatGPT are excellent at writing commit messages. Teach your juniors: after staging their changes, run* `git diff --cached` *and paste it into Copilot Chat with "Write a conventional commit message for these changes." This habit alone will clean up your git history significantly.*

PR Size Guidelines

Large PRs are a code quality problem. They're hard to review, hard to reason about, and AI tools tend to make them larger (because generating code is cheap). Set a soft limit:

- **Under 400 lines changed:** Ideal
- **400–800 lines:** Okay with good description
- **800+ lines:** Requires justification. Break it up if possible.

If a PR is large because it's a refactor with many small changes, use a detailed PR description and walk your reviewer through it. If it's large because the feature is genuinely big, break it into stacked PRs.

---

2. Code Review Process

The Golden Rule

> ***Every line that ships was approved by a human who understood it.***

This applies doubly to AI-generated code. Copilot writes code that looks correct. It compiles. It even has reasonable variable names. But it doesn't know your business logic, your edge cases, your data constraints, or your safety requirements. That's the reviewer's job.

PR Templates

Set up a PR template so every pull request answers the same questions. Create this file in your repo:

`.github/pull_request_template.md`

```
## Summary
<!-- What does this PR do? One to three sentences. Be specific. -->

## Changes Made
<!-- Bullet list of what changed -->
-
-

## How to Test
<!-- Step-by-step instructions to verify this works -->
1.
2.

## Screenshots / Demo
<!-- If UI changes, include a screenshot or Loom link -->

## Checklist

### General
- [ ] I have tested this locally
- [ ] I have written or updated tests for this change
- [ ] I have updated documentation if needed
- [ ] The PR is under 400 lines (or I've justified why it's larger)
- [ ] Branch is up to date with main

### Code Quality
- [ ] No hardcoded secrets, credentials, or API keys
- [ ] No debug logs, `console.log`, or `print` statements left in
- [ ] No TODO comments that should be tickets
- [ ] Variable and function names are clear without needing a comment to explain them
- [ ] Error cases are handled, not ignored

### AI-Generated Code
```

```
- [ ] I can explain every section of this code (including AI-assisted sections)
- [ ] I have reviewed AI suggestions critically and not merged blindly
- [ ] Edge cases have been considered, not just the happy path
- [ ] AI-generated logic has been verified against business requirements

### Safety-Critical (if applicable)
- [ ] This change has no effect on safety-critical paths, OR:
- [ ] I have tagged this PR `safety-critical` and requested senior review
- [ ] I have added/updated integration tests for affected safety logic
- [ ] I have documented the safety implications in this description
```

Review Roles: Human + AI

Modern code review is a collaboration between human reviewers and AI tooling. Here's how to structure it:

AI Review (automated — happens first): - Linting and formatting checks (ESLint, Prettier, Black, Flake8) - Static analysis (SonarQube, CodeClimate, or GitHub's built-in code scanning) - Security scanning (Dependabot, Snyk, GitHub Advanced Security) - Test coverage reports - GitHub Copilot code review (GitHub's PR review feature, if enabled)

Human Review (happens second, with AI context): - Business logic correctness - Architecture and design decisions - Naming and readability - Edge case coverage - Safety implications - Team knowledge transfer (does the reviewer understand what's being built?)

The reviewer's job is NOT to: - Re-run the linter (CI does that) - Manually check imports (the linter catches this) - Count lines (GitHub shows you)

The reviewer's job IS to: - Understand the change completely before approving - Ask "what happens when X is null?" or "what if the API returns an error here?" - Ensure the PR description is accurate - Catch logic errors that tests didn't cover - Make sure the code matches the intended design

Review Response Norms

Define what comments mean on your team. Left undefined, this creates friction.

Label	Meaning
Blocker / ❌	Must be fixed before merge
Suggestion / 💡	Take it or leave it — but acknowledge it
Nit	Tiny style thing, won't block merge
Question / ❓	Not blocking — just wants to understand
FYI	For awareness, no action needed

Example usage:

```
💡 Suggestion: You could extract this into a helper function — it's used in three places now.

❌ Blocker: This will crash if `user` is None. Need a null check here.

Nit: Prefer `is_valid` over `valid` for boolean variable names.
```

Review SLA

Set expectations around turnaround time:

- **Standard PR:** Review within 1 business day
- **Hotfix / unblocking:** Review within 2 hours (ping the reviewer directly)
- **Large PR (800+ lines):** Reviewer may request a walkthrough call

Reviews that sit for more than a day become stale, create merge conflicts, and kill developer momentum. Make this a cultural priority.

Pair Review for Juniors

For the first 90 days, consider requiring a junior's PRs to be reviewed by a senior (or yourself). This isn't distrust — it's investment. The fastest way to level up a junior is immediate, specific feedback on their actual code.

After 90 days, adjust based on demonstrated judgment. Some juniors are ready to review other juniors' code at 60 days. Some need more time. Be honest.

---

3. Coding Standards

Why Standards Beat "Best Practices"

"Best practices" are general. Standards are specific. "Write clean code" is a best practice. "All functions must have a docstring that includes a description, parameters, and return type" is a standard.

Standards are enforceable. Best practices are aspirational. For a junior-heavy team, standards win.

The goal is to make the right way the easy way. Automate what can be automated. Review what requires judgment. Argue about nothing that a linter can decide.

Python Standards (Example)

Adjust for your stack. The principles are universal.

Formatter: Black
No configuration debates. Black makes the decisions. Line length: 88 characters (Black's default). Done.

```
pip install black
black --check .     # in CI
black .             # to auto-fix
```

Linter: Flake8 + Ruff
Ruff is faster and handles most of what Flake8 + isort + pep8 used to require separately.

```
# pyproject.toml
[tool.ruff]
line-length = 88
select = ["E", "F", "W", "I", "N", "UP"]
ignore = ["E501"]  # Black handles line length

[tool.ruff.per-file-ignores]
"tests/*" = ["S101"]  # Allow assert in tests
```

Type hints: Required for all new code

```
# BAD — no type hints
def calculate_score(video, weights):
    return sum(weights[k] * video[k] for k in weights)

# GOOD — explicit types
def calculate_score(video: VideoAnalysis, weights: dict[str, float]) -> float:
    """Calculate weighted score for a video based on analysis results.

    Args:
        video: The analyzed video object containing scored dimensions.
        weights: A mapping of dimension names to their weight multipliers.

    Returns:
        A float between 0.0 and 1.0 representing the weighted overall score.

    Raises:
        KeyError: If a weight references a dimension not present in the video analysis.
    """
    return sum(weights[k] * getattr(video, k) for k in weights)
```

Type hints aren't just documentation — they enable static analysis tools to catch bugs before runtime.

Static type checking: mypy

```
mypy --strict src/
```

Run this in CI. Catch type errors before they reach production.

Naming Conventions

Clear naming is the single highest-leverage thing you can do for code quality. AI-generated code is often generic. Your reviewers should push back on generic names.

Context	Convention	Example
Variables	`snake_case`	`video_score`, `user_id`
Functions	`snake_case`, verb-first	`calculate_score()`, `get_user()`
Classes	`PascalCase`	`VideoAnalysis`, `ScoringEngine`
Constants	`SCREAMING_SNAKE_CASE`	`MAX_RETRY_COUNT`, `DEFAULT_THRESHOLD`
Booleans	`is_`, `has_`, `can_`, `should_` prefix	`is_valid`, `has_error`, `can_submit`
Private	Leading underscore	`_internal_helper()`
Files/modules	`snake_case`	`video_scoring.py`, `user_auth.py`

Names to reject in review: - `data`, `info`, `stuff`, `temp`, `result` (too generic) - `x`, `y`, `z` (outside of math/coordinates context) - `val`, `obj`, `thing` (almost always replaceable) - `manager`, `handler`, `processor` (usually a sign of unclear responsibilities — what does it manage/handle/process?)

AI-specific naming note: Copilot often suggests `result` as a variable name for the output of a function. Push back on this. Name the variable after what it *is*: `scored_video`, `validated_user`, `parsed_config`.

Documentation Requirements

Docstrings: Required for all public functions, classes, and modules. Use Google-style format:

```
def evaluate_driver(
    video_id: str,
    scoring_config: ScoringConfig,
    strict_mode: bool = False
) -> EvaluationResult:
    """Evaluate a driver's performance from a recorded video session.

    Runs the full scoring pipeline against the provided video, applying
    the weights and thresholds defined in the scoring config.

    Args:
        video_id: The unique identifier for the video in cloud storage.
        scoring_config: Configuration object defining scoring weights,
            thresholds, and evaluation criteria.
        strict_mode: If True, any failed safety check results in a
            hard failure rather than a weighted penalty. Defaults to False.

    Returns:
        An EvaluationResult containing the overall score, per-dimension
        scores, flags raised, and metadata about the scoring run.

    Raises:
        VideoNotFoundError: If the video_id does not exist in storage.
        ScoringConfigError: If the scoring_config is invalid or incomplete.
        InsufficientVideoError: If the video is too short to evaluate.

    Example:
        >>> config = ScoringConfig.from_yaml("configs/standard.yaml")
        >>> result = evaluate_driver("vid_abc123", config)
        >>> print(result.overall_score)
        0.87
    """
```

README requirements: Every repository and major module needs a README that covers: - What it does (1–2 sentences) - How to set it up locally - How to run tests - Key configuration options - Architecture overview (even a simple bullet list)

Architecture Decision Records (ADRs): When the team makes a significant technical decision, document it. Keep these in `/docs/decisions/`. Use a simple format:

```
# ADR-001: Use PostgreSQL for primary data store

**Status:** Accepted
**Date:** 2026-01-15
**Decision:** Use PostgreSQL over MongoDB for the primary application database.

**Context:**
Our data has strong relational structure (users → evaluations → videos → scores).
We need ACID guarantees for evaluation records.

**Decision:**
PostgreSQL, hosted on AWS RDS. ORM: SQLAlchemy.

**Consequences:**
- Schema migrations required (handled by Alembic)
- Better query performance for complex joins
- Team needs to learn SQL if they haven't yet
```

These are invaluable when onboarding new devs, when you're debugging 6 months later, and when investors ask "why did you build it this way?"

JavaScript/TypeScript Standards

For TypeScript-heavy teams, the same principles apply:

Formatter: Prettier
Linter: ESLint with `@typescript-eslint`
Types: TypeScript strict mode — always

```
// tsconfig.json
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true
  }
}
```

Never use `any`. If Copilot suggests `any`, reject it. Find the right type. If the right type is genuinely unclear, use `unknown` and narrow it. `any` is a backdoor that bypasses the entire type system.

---

4. AI-in-the-Loop Workflow

The 70/30 Rule

Here's the most important mental model for AI-augmented development:

> ***AI handles ~70% of the work. Your developers own the 30% that requires judgment, domain expertise, and architectural thinking.***

The 70% AI handles well: - Boilerplate code (CRUD operations, API endpoints, form validation) - Unit tests for well-defined functions - Docstrings and comments - Error handling patterns - Type annotations - Refactoring repetitive code - Converting between data formats - SQL queries for standard operations - Regular expressions - Commit messages and PR descriptions

The 30% humans own: - Architecture decisions (what to build, not just how) - Business logic (what the system should *do*) - Edge case identification (what could go wrong?) - Safety-critical path review - Code that touches security, authentication, or authorization - Any code whose failure would have real-world consequences - Technical debt trade-off decisions - Debugging subtle, context-dependent bugs

The goal isn't to minimize AI use. It's to be *intentional* about it.

When to Use AI Tools

Green light — use AI freely:

- Writing a new CRUD endpoint you've written dozens of times before
- Generating unit tests for a function you just wrote
- Converting a data structure from one format to another
- Writing the docstring for a function you just finished
- Asking "what's the idiomatic Python way to do X?"
- Generating boilerplate (Pydantic models, SQLAlchemy models, pytest fixtures)
- Refactoring: "take this function and split it into smaller pieces"
- Debugging: "here's my traceback and my code — what's wrong?"

Yellow light — use AI, but review carefully:

- Implementing business logic for the first time
- Writing code that touches the database schema
- Authentication and session handling
- Background jobs and async processing
- Any code that uses third-party APIs

Red light — do not use AI as primary author:

- Security-critical logic (permissions, access control, data encryption)
- Core safety-critical algorithms (covered in Chapter 6)
- Database migrations
- Code that processes PII or sensitive data
- Anything you don't understand well enough to explain line-by-line

The red light doesn't mean "never use AI here." It means: don't let Copilot write it from scratch and ship it. Use AI for reference, for explaining concepts, for checking your work — but the primary author should be a human who owns the logic.

How to Review AI-Generated Code

When a developer submits a PR with AI-generated code (and you should assume most PRs have some), the review process should be more rigorous, not less.

The CARE framework for reviewing AI code:

C — Correctness
Does it actually do what it's supposed to do? AI is excellent at writing code that *looks* correct. Run it. Test edge cases. Don't just read it.

```
# Copilot might generate this — looks fine at first glance:
def get_user_by_email(email: str) -> User:
    return db.query(User).filter(User.email == email).first()

# What happens if email is None? What if there are two users with the same email?
# What does the caller do with None if no user is found?
# These questions aren't answered by the code.
```

A — Appropriateness
Is this the right approach for *your* system? AI doesn't know your architecture, your performance requirements, or your team's conventions. It will generate valid code that may not fit.

R — Robustness
AI tends to generate the happy path. Ask: what if the input is null? What if the network call fails? What if the list is empty? What if the file doesn't exist?

E — Explainability
Can the developer who submitted it explain every line? If not, the code should not be merged. This is non-negotiable. "Copilot wrote it" is not a valid explanation of what code does.

Copilot Best Practices for Developers

Teach your developers these habits:

1. Write intent first, let AI fill in implementation

```
def calculate_driver_score(evaluation: DriverEvaluation) -> DriverScore:
    """
    Calculate final driver score from an evaluation.

    Weight the dimensions by importance:
    - Safety behaviors: 40%
    - Defensive driving: 30%
    - Vehicle control: 20%
    - Route efficiency: 10%

    Return a DriverScore with overall score and per-dimension breakdown.
    """
    # Copilot will now generate much better code because context is clear
```

2. Use Copilot Chat for understanding, not just generation

Teach your juniors these prompts: - `/explain` — "Explain what this function does line by line" - `/tests` — "Write unit tests for this function including edge cases" - `/fix` — "Here's the error I'm getting — what's wrong?" - "What are the failure modes for this code?" - "What security issues could this code have?"

3. Review suggestions before accepting

Never accept an entire multi-line suggestion without reading it. Use partial acceptance: - Accept a single word: `Ctrl+Right` - Accept a line: `Ctrl+Enter` then choose - Read the full suggestion before committing

4. Don't use AI for copy-pasting without understanding

The most dangerous junior developer pattern: accept Copilot's suggestion, see that it runs without errors, submit the PR. If the developer can't explain the code, the code doesn't get merged.

Prompting Standards

Good AI output starts with good input. Establish team standards for how to prompt:

Include context in comments:

```
# Parse the CSV export from the fleet management system.
# Format: date,driver_id,route_id,video_url,duration_seconds
# driver_id can be null for unassigned vehicles.
# duration_seconds is always positive but may be 0 for failed recordings.
def parse_fleet_csv(file_path: str) -> list[FleetRecord]:
```

Name things well — it's your best prompt: Copilot uses your variable names, function names, and file names as context. A function called `process_data()` will get worse suggestions than `validate_and_normalize_driver_score()`.

Ask for tests in the same session:
After Copilot helps write a function, immediately use Copilot Chat: "Write pytest tests for this function, including tests for null inputs and edge cases." Generate the tests while the context is warm.

The "Explain It Back" Rule

Before any AI-generated code merges, the developer must be able to:

1. Explain what the code does in plain English
2. Identify the inputs, outputs, and side effects
3. Describe what happens in at least three edge cases
4. Explain why this approach was chosen over alternatives

If they can't do this, they need to spend more time understanding the code before it ships. This is a teachable moment, not a punishment.

---

5. Onboarding New Developers

Week 1: Foundation Before Code

The worst thing you can do with a new junior developer is throw them at tickets on day one. They'll copy-paste code they don't understand, and Copilot will help them do it faster. This is not a win.

Week 1 is about understanding — the codebase, the domain, and the workflow.

Day 1–2: Environment & Culture

- [ ] Set up their development environment using the setup guide (Chapter 7)
- [ ] Walk through the entire tech stack — not to understand it all, but to get the map
- [ ] Read the README and top-level architecture docs
- [ ] Review the ADR log (they won't understand all of it — that's fine)
- [ ] Set up their GitHub Copilot and VS Code extensions
- [ ] Do the first commit: update their name in the contributors file (ensures environment is working)
- [ ] Read this playbook. All of it.

Day 3–5: Read First, Write Second

- [ ] Read 5–10 recent merged PRs — focus on the review comments, not just the code
- [ ] Shadow a senior developer or founder during a code review session
- [ ] Read 3 recent ADRs and ask questions about the decisions
- [ ] Explore the test suite — what's tested? What's not? Why?
- [ ] Set up local test database and get all tests passing

Buddy system: Assign every new hire a "code buddy" — a more experienced person they can ask questions without feeling embarrassed. Junior developers will sit on a question for an hour rather than ask a "dumb question." The buddy system removes that barrier.

Week 2–4: First Contributions

Structure their first tickets deliberately:

Tier 1 (Week 2): Zero-stakes contributions - Documentation improvements - Adding missing docstrings to existing functions - Writing tests for untested code - Fixing linting warnings

These contributions are real, valuable, and safe. They also force the new hire to read and understand existing code.

Tier 2 (Week 3): Guided feature work - Small, isolated bug fixes - Adding a new API endpoint that mirrors an existing one - Extending an existing model with a new field

First PR gets a thorough, teaching-focused review. The goal isn't just to catch problems — it's to explain *why* something is done differently and build judgment.

Tier 3 (Week 4): Independent feature work - Small standalone features with full PR review process - Must complete the full PR checklist independently - Reviewer asks them to explain one AI-generated section

The AI Ramp-Up Path

New junior devs are often already comfortable with Copilot from their bootcamp or side projects. The goal isn't to teach them to use it — it's to teach them *judgment* about when and how to use it.

Week 1–2: AI-off exercises
Ask them to write one feature without Copilot. Turn it off. See if they can reason through the problem. This isn't hazing — it's building the baseline understanding that makes AI use effective. You can't evaluate AI suggestions if you don't know what the code should look like without AI.

Week 3: AI-on, commentary required
When they use Copilot, they annotate: "Copilot suggested this. I changed X because Y. I verified it handles Z edge case."

Week 4+: Full workflow
AI tools on. But the first review of every Copilot-heavy PR includes the "Explain It Back" exercise.

Code Review as Teaching

For junior developers, code reviews are the primary source of technical mentorship. Treat them that way.

Good review comment (teaches):

```
❌ Blocker: This will throw a `KeyError` if `video_id` isn't in the dict.

Instead, use `dict.get(key, default)` which returns None (or your default)
if the key doesn't exist. This is the idiomatic Python pattern for optional dict access.

dict.get(video_id) returns None if video_id isn't present.
dict.get(video_id, []) returns an empty list as default.
```

Bad review comment (doesn't teach):

```
❌ Blocker: This will break.
```

The extra 30 seconds to explain *why* pays back in compounding improvement. A junior who understands why something is wrong won't make the same mistake 10 times.

Measuring Progress, Not Presence

Don't measure junior developers by hours logged or lines of code. Those metrics are meaningless with AI tools — Copilot inflates line count trivially.

Measure instead: - Tickets closed per sprint (features shipped, bugs fixed) - Review quality improvement over time (are their PRs getting cleaner?) - Bug introduction rate (how often does their code create new bugs?) - Independence — are they asking fewer clarifying questions over time? - Code review quality — are they catching real issues when reviewing others?

Review these metrics monthly in 1:1s. Junior developers who are improving but struggling deserve support and clarity. Junior developers who plateau at a low level despite support need an honest conversation.

---

6. Safety-Critical Considerations

What "Safety-Critical" Means Here

In this playbook, "safety-critical" means code where a bug could: - Cause harm to a person (physical, financial, reputational) - Produce an incorrect outcome in a consequential decision - Fail silently in a way that affects real-world results

This includes but isn't limited to: - Driver evaluation and scoring systems - Medical record handling - Financial calculations and reporting - Educational assessment systems - Access control for sensitive data - Automated decisions that affect people's opportunities or outcomes

The coding standards in Chapter 3 apply everywhere. The standards in this chapter apply *on top* of those, for safety-critical code paths specifically.

The Safety-Critical Classification System

Not all code is equal. Create a classification system your team understands:

🟢 **Standard code:**
Low or no real-world impact if wrong. Can be reviewed and merged with standard process.

Examples: UI changes, admin dashboard, internal tooling, logging improvements

🟡 **Sensitive code:**
Could affect user experience or data integrity. Requires careful review but not extra gates.

Examples: API endpoints that read/write user data, email notifications, data exports

🔴 **Safety-critical code:**
A bug could cause incorrect outcomes that affect people. Requires elevated review process.

Examples: Scoring algorithms, evaluation logic, access control, any code that produces a decision used by a human in a consequential context

Label PRs appropriately. Use GitHub labels. Make it visible.

Extra Gates for Safety-Critical PRs

Standard PR process (Chapter 2) + these additional requirements:

1. Senior/founder review required
Safety-critical PRs must be approved by a senior developer or the technical founder. No exceptions. A junior developer can write safety-critical code — they cannot be the sole reviewer of it.

2. Two-reviewer minimum
At least two people must approve. A second set of eyes is not overhead — it's the minimum viable safety check for consequential code.

3. Explicit edge case documentation
The PR description must include a section: "Edge Cases Considered"

```
## Edge Cases Considered

- **Zero-frame video:** Returns `EvaluationError` with code `INSUFFICIENT_FOOTAGE`
- **Null driver ID:** Raises `ValidationError` before scoring begins
- **Score weights don't sum to 1.0:** Normalized automatically; logged as warning
- **All dimensions score 0:** Valid result, returned as-is (not treated as error)
- **Video duration < 30 seconds:** Returns `EvaluationError` with code `VIDEO_TOO_SHORT`
```

4. Integration tests required
Unit tests are necessary but not sufficient for safety-critical code. You need integration tests that test the full pipeline:

```
# tests/integration/test_scoring_pipeline.py

class TestScoringPipelineSafetyEdgeCases:
    """Integration tests for safety-critical scoring pipeline paths."""

    def test_short_video_returns_error_not_zero_score(self, db_session):
        """Ensure short videos are rejected, not scored as zero."""
        video = create_test_video(duration_seconds=15)
        result = scoring_pipeline.run(video.id, ScoringConfig.default())

        assert result.status == EvaluationStatus.ERROR
        assert result.error_code == "VIDEO_TOO_SHORT"
        assert result.overall_score is None  # Not zero — None. Important distinction.

    def test_score_is_deterministic(self, db_session):
        """Same video, same config must always produce same score."""
        video = create_test_video()
        config = ScoringConfig.default()

        result_1 = scoring_pipeline.run(video.id, config)
        result_2 = scoring_pipeline.run(video.id, config)
        assert result_1.overall_score == result_2.overall_score
        assert result_1.dimension_scores == result_2.dimension_scores
```

5. Staging validation
Safety-critical changes must pass in staging with real (or realistic) data before production deploy. Don't use production as your test environment for code that affects outcomes.

AI Usage Rules for Safety-Critical Code

AI is a tool with known failure modes. In safety-critical paths, those failure modes are unacceptable:

AI must not be the primary author of: - Scoring or evaluation algorithms - Access control logic (who can see/change what) - Data validation that affects downstream safety decisions - Threshold or weight configuration parsing - Any code path that produces a binary pass/fail on a person

AI may assist with: - Writing tests for safety-critical code (after a human has written the logic) - Generating boilerplate wrappers around safety-critical core logic - Documentation and comments - Explaining how the existing code works (for review purposes) - Identifying potential edge cases: "What could go wrong with this function?"

The review question for safety-critical AI code:
"If this code produced an incorrect result, how would we know? And how quickly?"

If the answer is "we'd know when a person complains" — that's not good enough. Build observability in from the start.

Observability as a Safety Requirement

For safety-critical systems, logging and monitoring aren't nice-to-haves. They're requirements.

Minimum observability for safety-critical paths:

```
import structlog
from datetime import datetime

logger = structlog.get_logger(__name__)

def evaluate_driver(video_id: str, config: ScoringConfig) -> EvaluationResult:
    log = logger.bind(
        video_id=video_id,
        config_version=config.version,
        evaluation_timestamp=datetime.utcnow().isoformat()
    )

    log.info("evaluation.started")

    try:
        result = _run_scoring_pipeline(video_id, config)
```

```
            log.info(
                "evaluation.completed",
                overall_score=result.overall_score,
                dimension_count=len(result.dimensions),
                flags_raised=len(result.safety_flags),
            )

            return result

        except InsufficientVideoError as e:
            log.warning("evaluation.failed.insufficient_video", error=str(e))
            raise

        except Exception as e:
            log.error("evaluation.failed.unexpected", error=str(e), exc_info=True)
            raise
```

This structured logging means you can answer: "What score did driver #1234 get on January 15th, and what config version was used?" — instantly. That's the standard to hold yourself to.

Version everything: - Scoring algorithm version - Config file version - Model weights (if using ML) - Any threshold that affects an outcome

When something is questioned, you need to know exactly what was running when.

Human-in-the-Loop for High-Stakes Decisions

For decisions with significant real-world consequences (employment, certification, legal, medical), automate the *evidence gathering* — not the *decision*.

The system produces: a score, a breakdown, flagged behaviors, a recommendation.
A human produces: the decision, with the system's output as input.

This isn't just an ethical position — it's a liability position. Build systems where human judgment is explicitly in the chain for high-stakes outcomes.

---

7. Tools & Setup Checklist

Development Environment Setup

Use this checklist when onboarding a new developer or setting up a new machine.

VS Code Configuration

```
# Install VS Code extensions
code --install-extension github.copilot
code --install-extension github.copilot-chat
code --install-extension ms-python.python
code --install-extension ms-python.vscode-pylance
code --install-extension charliermarsh.ruff
code --install-extension ms-python.black-formatter
code --install-extension eamodio.gitlens
code --install-extension streetsidesoftware.code-spell-checker
code --install-extension usernamehw.errorlens
```

`.vscode/settings.json` (commit this to the repo):

```
{
  "editor.formatOnSave": true,
  "editor.codeActionsOnSave": {
    "source.fixAll.ruff": "explicit",
    "source.organizeImports.ruff": "explicit"
  },
  "[python]": {
    "editor.defaultFormatter": "ms-python.black-formatter"
  },
  "python.typeCheckingMode": "strict",
  "python.analysis.typeCheckingMode": "strict",
  "editor.rulers": [88],
  "editor.tabSize": 4,
  "files.trimTrailingWhitespace": true,
  "files.insertFinalNewline": true,
  "github.copilot.enable": {
    "*": true,
    "markdown": true,
    "yaml": false
  }
}
```

> ***Note on YAML and Copilot:*** *Disable Copilot for YAML files. Copilot's suggestions for config files (especially CI/CD and infrastructure) can introduce subtle misconfiguration. Always write config by hand.*

Git Configuration

```
# Set up global git config
git config --global user.name "First Last"
git config --global user.email "name@company.com"
git config --global pull.rebase true
git config --global fetch.prune true

# Set default branch name to main
git config --global init.defaultBranch main
```

```
# Better diff output
git config --global diff.colorMoved zebra
```

`.gitignore` (Python project template):

```
# Python
__pycache__/
*.py[cod]
*.so
.Python
.venv/
venv/
env/
.env
*.egg-info/
dist/
build/

# Testing
.pytest_cache/
.coverage
coverage.xml
htmlcov/

# Type checking
.mypy_cache/

# Editors
.vscode/
!.vscode/settings.json
!.vscode/extensions.json
.idea/

# OS
.DS_Store
Thumbs.db

# Secrets — never commit these
.env
.env.local
*.pem
*.key
secrets/
```

Python Environment Setup

```
# Install Python version manager (pyenv)
curl https://pyenv.run | bash
```

```
# Install target Python version
pyenv install 3.12.3
pyenv local 3.12.3

# Create virtual environment
python -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate

# Install dev dependencies
pip install -r requirements-dev.txt
```

`requirements-dev.txt` template:

```
# Core
-r requirements.txt

# Testing
pytest>=7.4
pytest-cov>=4.1
pytest-asyncio>=0.21
factory-boy>=3.3

# Type checking
mypy>=1.6

# Linting & formatting
ruff>=0.1
black>=23.0

# Development utilities
ipython>=8.0
python-dotenv>=1.0
structlog>=23.0
```

GitHub Repository Setup

Repository setup checklist:

- [ ] Main branch created
- [ ] Branch protection rules configured (see Chapter 1)
- [ ] PR template created (`.github/pull_request_template.md`) (see Chapter 2)
- [ ] `.gitignore` committed
- [ ] `.vscode/settings.json` committed
- [ ] `CONTRIBUTING.md` created
- [ ] `README.md` with setup instructions
- [ ] GitHub Actions CI pipeline set up

- [ ] Dependabot enabled for dependency updates
- [ ] Required labels created: `safety-critical`, `breaking-change`, `do-not-merge`
- [ ] Team members added with appropriate permissions
- [ ] Copilot for Business enabled at the organization level

CI/CD Pipeline

A basic but complete GitHub Actions pipeline:

`.github/workflows/ci.yml`:

```yaml
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  quality:
    name: Code Quality
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"
          cache: pip

      - name: Install dependencies
        run: pip install -r requirements-dev.txt

      - name: Lint with Ruff
        run: ruff check .

      - name: Check formatting with Black
        run: black --check .

      - name: Type check with mypy
        run: mypy --strict src/

      - name: Run tests with coverage
        run: |
          pytest tests/ \
            --cov=src \
            --cov-report=xml \
            --cov-report=term-missing \
            --cov-fail-under=80

      - name: Upload coverage report
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage.xml
```

Key CI rules: - CI must pass before a PR can merge (enforce via branch protection) - Tests must maintain >80% coverage (adjust based on your risk tolerance) - Type checking is non-negotiable — `mypy --strict` in CI - Formatting check, not auto-fix (auto-fix hides what changed)

Security Scanning

```
# Add to CI or as separate workflow
  security:
    name: Security Scan
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Run Bandit security scan
        run: |
          pip install bandit
          bandit -r src/ -ll  # Report medium and high severity

      - name: Check for secrets in code
        uses: trufflesecurity/trufflehog@main
        with:
          path: ./
          base: ${{ github.event.repository.default_branch }}
          head: HEAD
```

Dependabot configuration (`.github/dependabot.yml`):

```
version: 2
updates:
  - package-ecosystem: pip
    directory: "/"
    schedule:
      interval: weekly
    open-pull-requests-limit: 5
    labels:
      - "dependencies"
```

Copilot for Teams: Configuration and Policies

GitHub Copilot Business (vs. Individual) gives you: - Centralized license management - Policy controls (disable for certain repos or file types) - Audit logs of Copilot usage - IP indemnification for enterprise

Recommended organization policies:

```
Copilot Business Settings:
  ✓ Allow Copilot for all members
  ✓ Enable Copilot Chat in IDE
  ✓ Enable Copilot in GitHub.com (PR descriptions, code explanations)
  ✗ Disable: suggestions matching public code (reduces IP risk)
```

The "suggestions matching public code" setting prevents Copilot from suggesting verbatim code from public repositories. For safety-critical software, this reduces the risk of inadvertently importing bugs or insecure patterns from public code.

Complete Tools & Stack Checklist

Use this as a day-one setup guide:

Developer Machine

- [ ] Python 3.11+ (via pyenv)
- [ ] Git 2.40+
- [ ] VS Code (latest)
- [ ] GitHub Copilot extension + Copilot Chat
- [ ] GitLens extension
- [ ] Ruff extension
- [ ] Black formatter extension
- [ ] Pylance (Python language server)
- [ ] ErrorLens (inline error display)
- [ ] Node.js 20+ (for frontend/tooling)

GitHub Repository

- [ ] Branch protection on `main`
- [ ] PR template in `.github/`
- [ ] CI/CD GitHub Actions workflow
- [ ] Dependabot enabled
- [ ] Code scanning enabled
- [ ] Secret scanning enabled
- [ ] Required labels created

Project Configuration (committed to repo)

- [ ] `.vscode/settings.json`
- [ ] `.gitignore`
- [ ] `pyproject.toml` (Ruff/Black/mypy config)
- [ ] `requirements.txt` + `requirements-dev.txt`
- [ ] `pytest.ini` or `[tool.pytest.ini_options]` in pyproject.toml
- [ ] `.github/workflows/ci.yml`
- [ ] `.github/dependabot.yml`
- [ ] `CONTRIBUTING.md`

Team Accounts & Access

- [ ] GitHub organization created, team invited
- [ ] Copilot Business licenses assigned
- [ ] Shared secrets in GitHub Secrets (never in code)
- [ ] Staging environment credentials distributed securely
- [ ] CI/CD secrets configured

Appendix A: Quick Reference — The Non-Negotiables

These aren't suggestions. These are the rules that apply from day one, no exceptions:

1. **No direct pushes to `main`** — every change goes through a PR
2. **Every PR gets a human review** — AI review augments, never replaces
3. **No AI-generated code without developer understanding** — "Copilot wrote it" is not an explanation
4. **Safety-critical PRs get two reviews** — senior reviewer required
5. **All tests must pass in CI before merge** — no "I'll fix it after"
6. **No secrets in code** — use environment variables and GitHub Secrets
7. **Linting and formatting enforced by CI** — not by human reviewers
8. **Type hints required** on all new code — no `any`

Appendix B: Recommended Reading & Resources

On Software Engineering Fundamentals: - *A Philosophy of Software Design* — John Ousterhout - *The Pragmatic Programmer* — Hunt & Thomas - *Accelerate* — Forsgren, Humble, Kim (on what actually predicts software delivery performance)

On AI-Augmented Development: - GitHub Copilot documentation and best practices guide - *Prompt Engineering for Developers* — deeplearning.ai (free course) - OpenAI's usage policies for code generation (understand the tool's limitations)

On Safety-Critical Systems: - *Working in Public: The Making and Maintenance of Open Source Software* — Nadia Eghbal (on software trust and reliability) - NIST Software Security Guidelines

---

Appendix C: PR Template — Final Version

Save this as `.github/pull_request_template.md` in your repository.

```
## Summary
<!-- What does this PR do? One to three sentences. Be specific. -->

## Changes Made
<!-- Bullet list of what changed and why -->
-
-

## How to Test
<!-- Step-by-step testing instructions -->
1.
2.

## Screenshots / Demo
<!-- UI changes: include a screenshot or Loom link -->
<!-- API changes: include example request/response -->

## Edge Cases Considered
<!-- List cases you thought about, even if you determined they weren't issues -->
-

---

## Checklist

### Required for All PRs
- [ ] I have tested this change locally
- [ ] I have written or updated tests for this change
- [ ] All existing tests pass
- [ ] The PR is under 400 lines (or I've explained why it's larger)
- [ ] Branch is up to date with main

### Code Quality
- [ ] No hardcoded secrets, credentials, or API keys
- [ ] No debug logs or print statements left in production code
- [ ] No TODOs that should be tracked as tickets
- [ ] Error cases are handled, not silently swallowed
- [ ] Variable and function names clearly describe their purpose

### Documentation
- [ ] New public functions have docstrings
- [ ] README updated if setup/usage changed
- [ ] Architecture decisions documented if applicable

### AI-Generated Code
- [ ] I can explain every section of this code in plain English
- [ ] I reviewed AI suggestions critically — not accepted blindly
- [ ] Edge cases are covered, not just the happy path
- [ ] AI-generated logic verified against actual requirements

### Safety-Critical (complete if applicable)
- [ ] This change does NOT affect safety-critical paths → standard review applies
- [ ] This change DOES affect safety-critical paths:
  - [ ] Tagged with `safety-critical` label
  - [ ] Senior reviewer requested
  - [ ] Two approvals required before merge
  - [ ] Integration tests added/updated for affected paths
  - [ ] Observability/logging confirmed for affected code paths
  - [ ] Edge cases documented in "Edge Cases Considered" section above
```

Obed Industries — AI-powered business strategy and implementation

This playbook is a living document. Review and update it quarterly as your team grows and your tools evolve. The best process is the one your team actually follows.

Document version: 1.0
Last updated: March 2026
Maintained by: Obed Industries
