Heuristic Weighting

🔹 Step 1: Define Base Weights

Let's say you assign **default weights** for categories (summing to 100%):

- Code Generation → 50%
- Code Quality → 25%
- Code Improvement → 15%
- Code Understanding → 10%

🔹 Step 2: Normalize Weights for Selected Categories

When fewer categories are tagged, you **re-normalize only among those selected**.

👉 Example 1: If a ticket is tagged with [Code Generation, Quality]:

- From base weights → Gen = 50, Quality = 25.
- Total = 75.
- Normalize:
    - Gen = 50/75 = **66.7%**
    - Quality = 25/75 = **33.3%**

So if story points = 10, reduction = 25% (2.5 SP saved):

- Gen gets 1.67 SP saved.
- Quality gets 0.83 SP saved.

👉 Example 2: If a ticket is tagged with [Improvement, Understanding]:

- Base = 15 + 10 = 25.
- Normalize:
    - Improvement = 15/25 = **60%**
    - Understanding = 10/25 = **40%**

If story points = 8, reduction = 15% (1.2 SP saved):

- Improvement = 0.72 SP.
- Understanding = 0.48 SP.

👉 Example 3: If **all 4 categories** are selected → just use base weights as-is.

🔹 Step 3: Implementation Formula

If `SavedPoints = StoryPoints × Reduction%`

For each category:

```
CategorySavedSP = SavedPoints × (CategoryWeight / SumOfSelectedCategoryWeights)
```

✅ This way:

- You avoid equal splits.
- You bias attribution to categories that you believe **drive more time savings**.
- You keep results **consistent and defensible**.

Copilot Adoption & Efficiency Insights Report

1. Adoption by Copilot Category

- **What it tells us:**
  Distribution of usage across categories like code generation, refactoring, documentation, boilerplate, etc.

- **Insights:**

    - Heavy skew towards certain categories (e.g., boilerplate) suggests untapped potential in advanced areas (e.g., refactoring, test generation).

    - Some categories may be underutilized simply because devs don't know they exist.

- **Next steps:**

    - Provide **category-focused training** (e.g., "how to use Copilot for test cases").

    - Create **prompt libraries/templates** for underused categories.

    - Encourage developers to experiment with multiple categories.

- **Efficiency Mapping:**
  Expanding adoption from low-value categories (boilerplate) to high-value ones (test generation/refactoring) increases **time savings per ticket**.

---

3. Copilot Category vs. Time Reduced Category

- **What it tells us:**
  Shows which Copilot categories correlate with **significant vs. moderate vs. slight time savings**.

- **Insights:**
    - Code generation may consistently drive “significant” savings.
    - Documentation may be in “slightly reduced” → shows diminishing returns.
    - Some categories may show inconsistent gains → needs further study.
- **Next steps:**
    - Prioritize expanding use of categories with **high impact correlation** (e.g., code generation, test generation).
    - Reduce reliance on categories with **low impact** (e.g., documentation).
    - Track how categories shift over time with training.
- **Efficiency Mapping:**
  Shifting work towards high-impact categories maximizes **efficiency per usage hour**.

---

4. Split Across Time Reduced Category (Significant, Moderate, Slight, Not Feasible)

- **What it tells us:**
  Shows distribution of tickets by **level of time saved**.
- **Insights:**
    - If many tickets are “slight” → Copilot is being under-leveraged.
    - If large % are “not feasible” → Copilot applicability may be low or devs aren’t framing prompts correctly.
- **Next steps:**
    - Investigate why “not feasible” tickets exist (domain complexity, poor prompting, setup issues).
    - Share **prompting best practices** to move “slight” → “moderate/significant”.

    - Track feasibility % per quarter to measure maturity.

- **Efficiency Mapping:**
  Increasing share of “significant” time savings tickets → exponential impact on total **efficiency gain**.

---

5. Ticket Size vs. Copilot Category Count

- **What it tells us:**
  Shows if Copilot is being used more on **small, medium, or large tickets**, and in how many categories.

- **Insights:**

    - If adoption is skewed towards small tickets → missing efficiency in **large/complex tickets**.

    - If larger tickets use multiple categories → shows Copilot’s compound benefits.

- **Next steps:**

    - Encourage Copilot use in **medium/large tickets** where returns are higher.

    - Build guidelines: “Use at least 2 categories (gen + test) for large tickets.”

    - Measure efficiency per story point bucket.

- **Efficiency Mapping:**
  Shifting Copilot from **small → large tickets** increases **weighted efficiency gains**.

---

6. Total Efficiency Gain

- **What it tells us:**
  Aggregated productivity improvement from Copilot.

- **Insights:**

    - Quantifies ROI of adoption so far.
    - Trend over time shows whether efficiency is compounding or stagnating.
- **Next steps:**
    - Establish a **baseline benchmark** (e.g., 20% gain).
    - Track monthly/quarterly growth.
    - Set target efficiency goals for leadership visibility.
- **Efficiency Mapping:**
  Direct measure of ROI → ties adoption strategy to tangible outcomes.

---

7. Efficiency by Copilot Usage Category

- **What it tells us:**
  Shows which categories drive the most efficiency.
- **Insights:**
    - Code generation/test automation often leads in efficiency.
    - Some categories may have **low adoption + low efficiency** → candidates for deprioritization.
- **Next steps:**
    - Focus enablement on **categories with high efficiency + medium adoption** (growth potential).
    - Evaluate whether to **sunset or de-prioritize** low adoption/low efficiency ones.
- **Efficiency Mapping:**
  Better allocation of developer time across categories → maximized ROI.

---

8. Efficiency by Story Point

- **What it tells us:**
  Maps Copilot impact against ticket size (e.g., 1SP, 3SP, 5SP, 8SP).

- **Insights:**

  - Small tickets may show marginal benefit → Copilot more valuable in **5SP+ tickets**.

  - Large story points could demonstrate compounding savings.

- **Next steps:**

  - Prioritize Copilot use on **medium-to-large tickets**.

  - Create guidelines: “Copilot is mandatory for 5SP+ stories.”

- **Efficiency Mapping:**
  Focus on **story-point weighted efficiency** to drive bigger ROI.

---

9. Efficiency by Ticket Size

- **What it tells us:**
  Direct comparison of efficiency vs. ticket size.

- **Insights:**

  - Efficiency % may plateau beyond a certain ticket size → diminishing returns.

  - High adoption in small tickets but low efficiency → wasted effort.

- **Next steps:**

  - Encourage **balanced adoption** (not just small tickets).

  - Study where Copilot doesn’t scale well (very large tickets) → supplement with other techniques.

- **Efficiency Mapping:**
  Aligning Copilot use to **optimal ticket size range** boosts aggregate ROI.

Copilot Adoption & Efficiency —

Executive summary

- **Adoption is high (68%)**, and **overall efficiency gain is ~7%** at the app level.
- **Biggest ROI categories:** *Code Improvement (8%)* and *Code Generation (7%)*.
- **By size:** Medium & large tickets deliver **~7%** efficiency; small tickets lag at **~5%**.
- **Where time savings land today:** Mostly **slight/moderate** reductions; "significant" is only **~7% of issues**.

What the data tells us (with takeaways)

1) Adoption by App

- **Used:** 541 (68%) **Not used:** 267 (32%).
- **So what:** Copilot is embedded in day-to-day flow, but 1 in 3 issues still don't use it.
- **Next step:** Run a quick scan of "not used" issues to separate **not applicable** vs **missed opportunity** and target the latter with prompts/examples.

2) Adoption by Category

- **Share of issues:** Improvement **29%** > Generation **26%** > Quality **24%** > Understanding **20%**.
- **So what:** Team relies most on **Improvement**/**Generation**—the same areas with strongest ROI (see next section).
- **Next step:** Double-down enablement and prompt packs for **Improvement** & **Generation**; encourage secondary use (e.g., add *Testing/Quality* asks after generation).

3) Efficiency by Category (story-point weighted)

- **Improvement: 8% Generation: 7% Quality: 4% Understanding: 4%**.
- **So what:** Copilot pays off most when **writing or upgrading code**; returns are lower for quality/understanding-only tasks.
- **Next step:**
    - Make **Improvement** and **Generation** the "default Copilot lanes" for engineers.
    - For **Quality/Understanding**, supplement with structured checklists or automated review bots to convert "no reduction" cases into "slight/moderate".

4) Efficiency by Story Points

- **Peaks:** 13-point stories **11%**, and 5/8-point stories **8%**.

- **Low impact:** 1–3 SP **5–6%**; **6 SP** and **30 SP** currently **0%**.
- **So what: Mid-sized stories** are the Copilot sweet spot. Very small tasks are already fast; very large tasks need better decomposition/context.
- **Next step:** Encourage **splitting large stories**; set an expectation that **5–13 SP stories** use Copilot with multi-category prompts (gen + improve + tests).

5) Efficiency by Ticket Size (aggregated)

- **Medium: 7%**, **Large: 7%**, **Small: 5%** (by story-point weighting).
- **So what:** Despite higher adoption on larger work, **small tickets underperform** on ROI; medium/large can do even better with better prompts/context.
- **Next step:** Provide **prompt recipes** for medium/large tickets (e.g., "generate + refactor + create tests") and **skip low-value Copilot usage** on trivial small tickets.

6) Adoption × Time-reduction mix

- Counts: **Not used 264**, **Significant 56**, **Moderate 96**, **Slight 322**, **No reduction 67**. (That's ~**7% significant**, **12% moderate**, **40% slight**, **8% no reduction**, **32% not used**.)
- **So what:** Most Copilot-used issues land in **slight/moderate** buckets; **significant** is rare.
- **Next step (high-leverage):** Move slices of **slight** → **moderate** and **moderate** → **significant** with better prompting & context injection.

7) Usage by Ticket Size × Category (shares within each size)

- **Small:** "Not applicable" **35%** (i.e., many small issues skip Copilot).
- **Medium:** balanced use across categories (17–24%).
- **Large:** lowest "not applicable" (**9%**); **Improvement dominates (39%)**.
- **So what:** Engineers already reach for Copilot on larger work—good—but we're leaving easy wins on the table in small tickets where generation + tests can be quick wins.
- **Next step:** Where small tickets are repetitive (CRUD, stubs, tests), ship **one-click prompt templates** so devs get "instant" value.

Targeted actions (mapped to efficiency lift)

1. **Scale the winners (Generation & Improvement)**

- Action: Publish 6–8 **prompt recipes** (new module scaffolding, API handler + tests, refactor for readability/perf, migration fixups).

- Expected lift: even a **10% conversion of "slight" → "moderate"** within used issues adds **~0.6 pp** to average time saved among Copilot-used issues (and **~0.4 pp** across all issues).

*If "slight" → "significant" for the same 10%, the lift is **~1.2 pp** (used) / **0.8 pp** (all).*

2. **Decompose large/30-SP stories**

- Action: Policy: break 30-SP stories into **5–13 SP** sub-stories (where your data peaks at **8–11%** gains).

- Expected lift: converts zero-gain 30-SP work into **8–11%** territory.

3. **Raise the floor for small tickets**

- Action: For repetitive small tickets, standardize **"gen + tests"** micro-prompts (one launcher per repo).
- Expected lift: moving small from **5%** → **6–7%** can matter because they're numerous; this also frees attention for bigger tasks.

4. **Fix the bimodal "Quality" category**

- Your time distribution shows **Quality** has **31% significant and 31% no reduction**—i.e., inconsistent ROI.
- Action: Add **guardrails** (lint/fix prompts, static-analysis summaries, PR review checklists) so quality tasks don't fall to "no reduction".
- Expected lift: converting just **10% of "no reduction"** quality issues to **slight** adds measurable points.

5. **Shrink the "not used" pool with intent**

- Action: For the **32% not used**, only target the **applicable** subset (e.g., bugs, refactors, new endpoints).
- Expected lift: converting **10% of "not used"** to **moderate** (15%) adds roughly **+0.48 pp** to average time saved across all issues.

What to track next (to prove the lift)

- **Adoption × Impact funnel:** % Used → % Significant/Moderate/Slight (trend by sprint).
- **Category ROI:** Efficiency by category and by **multi-category usage** on a ticket.
- **Size ROI:** Efficiency by **SP buckets** (1,2,3,5,8,13,30) and **ticket size (small/medium/large)**.
- **"What-if" scorecard:** Show the effect of shifting portions of **slight** → **moderate** and **not used** → **moderate** each sprint (using the 25/15/5% rubric).

TL;DR leadership message

- We’re at **68% adoption** and **~7% efficiency** overall.
- Concentrate on **Code Improvement & Generation** and **mid-sized stories** to move efficiency fastest.
- Apply prompt packs, ticket decomposition, and QA guardrails to **lift “slight” to “moderate/significant.”**

🔹 1. Internal Codebase Complexity & Libraries

- **Challenge:** Copilot works best with well-known open-source patterns, but enterprise teams often use **custom internal libraries, frameworks, or DSLs**. Since Copilot has less context on these, suggestions are weaker.

- **Efficiency Impact:**
    - Developers spend time rejecting/refactoring irrelevant Copilot code.
    - Gains from repetitive/common UI or service layer code are low because internal frameworks differ.

👉 *Next Step:* Fine-tune Copilot on **internal codebase** or build **prompt templates** (e.g., "Write controller using our `BaseController` pattern") to help Copilot adapt.

🔹 3. Lack of Standardization in Coding Practices

- **Challenge:** If teams don't have consistent **coding guidelines or architecture patterns**, Copilot produces inconsistent code.

- **Efficiency Impact:**
    - More review cycles needed → productivity drops.
    - Junior engineers might rely too much, producing inconsistent PRs.

👉 *Next Step:* Establish **strong code standards** + enforce via **linters/PR checks**, so Copilot outputs are aligned.

🔹 4. Context Switching & Limited History

- **Challenge:** Copilot does not always understand **Jira ticket context**, **design docs**, or **commit history**.
- **Efficiency Impact:**
    - Developers spend time "re-explaining" context inside comments.
    - Suggestions miss business logic (e.g., "validation must match product rules").

👉 *Next Step:* Connect Jira + GitLab metadata into Copilot (through plugins/extensions), so AI suggestions are contextual.

🔹 5. Legacy Code & Monolithic Systems

- **Challenge:** Enterprises often have **legacy monoliths** with outdated patterns. Copilot is better with modern modular code.

- **Efficiency Impact:**
    - Developers spend more time fixing/refactoring AI output than writing directly.
    - Gains are seen only in greenfield or microservice areas, not old modules.

👉 *Next Step:* Prioritize **Copilot adoption on new projects/microservices** first.

🔹 7. Developer Mindset & Adoption Curve

- **Challenge:** Senior devs may resist Copilot ("I code faster without it"), juniors may over-rely.

- **Efficiency Impact:**
    - Uneven adoption → partial team gains.
    - Net productivity impact gets diluted.

👉 *Next Step:* Run **structured adoption pilots** + share internal **success stories** (ex: "UI tickets reduced by 30% dev time with Copilot").

✅ Efficiency-Specific Pain Points (Summary)

- Internal libraries not understood by Copilot → low gains.
- No Jira/GitLab context integration → Copilot lacks business logic awareness.
- Legacy code lowers suggestion quality → slows adoption.
- ROI not measured → leadership skeptical.
- Security/IP concerns → restrict usage.

> *"Our biggest blockers to efficiency are: Copilot not learning our internal libraries, lack of Jira/GitLab context, and limited adoption in legacy systems. If we fix these, efficiency gains could scale 2–3x beyond current levels."*

Here's a structured table version:

📊 Copilot Adoption & Efficiency Insights

Section	What it Tells Us	Insights	Next Steps	Efficiency Mapping
1. Adoption by Copilot Category	Distribution of usage across categories (generation, refactoring, docs, boilerplate).	Heavy skew towards boilerplate; advanced areas (refactoring, tests) underused; some categories unused due to lack of awareness.	Provide category-focused training; create prompt libraries/templates; encourage multi-category experimentation.	Moving usage from low-value (boilerplate) to high-value (tests/refactoring) drives higher time savings.
3. Copilot Category vs. Time Reduced	Correlation between category and time savings.	Code generation → significant savings; documentation → slight savings; some categories inconsistent.	Expand high-impact categories (gen/tests); reduce low-impact reliance (docs); track trends post-training.	Shifting work to high-impact categories maximizes efficiency per usage hour.
4. Split Across Time Reduced (Significant, Moderate,	Distribution of tickets by level of time saved.	Many "slight" → under-leverage; many "not feasible" → prompt/domain/setup issues.	Investigate "not feasible" cases; share prompting best practices; track feasibility % quarterly.	More "significant" tickets = exponential efficiency.

Slight, Not Feasible)				
5. Ticket Size vs. Copilot Category Count	Usage patterns across small/medium/large tickets and # of categories used.	Skew to small tickets → missing efficiency in large; larger tickets use multiple categories (compound benefit).	Encourage usage on medium/large tickets; guidelines: “Use ≥2 categories for large tickets”; measure efficiency per story point.	Moving Copilot from small → large tickets increases weighted efficiency gains.
6. Total Efficiency Gain	Aggregated productivity gain.	Quantifies ROI; trends show compounding vs stagnation.	Establish baseline (e.g., 20% gain); track quarterly; set targets for leadership.	Direct ROI measure linking adoption strategy to outcomes.
7. Efficiency by Usage Category	Which categories deliver most efficiency.	Code generation & test automation lead; some categories low adoption + low efficiency.	Focus on high-efficiency + medium adoption categories; de-prioritize low ROI ones.	Optimized allocation of dev time → maximized ROI.
8. Efficiency by Story Point	Impact by ticket size (SP buckets).	Small tickets marginal benefit; 5SP+ show compounding savings.	Prioritize Copilot for medium-large SP; “Copilot mandatory for 5SP+”.	Story-point weighted efficiency drives larger ROI.

9. Efficiency by Ticket Size	Efficiency vs ticket size (small/medium/large).	Efficiency plateaus for very large tickets; small tickets show wasted effort.	Balance adoption; study scalability issues; supplement very large with other methods.	Aligning usage with optimal ticket sizes boosts aggregate ROI.

📌 Executive Summary (Condensed)

Metric	Data	So What	Next Step
Adoption (App level)	68% used (541), 32% not used (267)	Copilot embedded in flow, but 1/3 issues untouched.	Scan “not used” → separate not applicable vs missed opportunity.
Adoption (Category)	Improvement 29%, Generation 26%, Quality 24%, Understanding 20%	Reliance on high ROI areas (Improvement/Gen).	Double-down enablement & prompt packs for Improvement & Gen; encourage Testing/Quality as add-ons.
Efficiency (Category)	Improvement 8%, Generation 7%, Quality 4%, Understanding 4%	ROI highest for writing/upgrading code; lower for docs/understanding.	Make Improvement/Gen default; add checklists/review bots for Quality/Understanding.
Efficiency (Story Points)	13SP: 11%, 5/8SP: 8%, 1–3SP: 5–6%, 30SP: 0%	Mid-sized stories are sweet spot; small too trivial, large too complex.	Decompose large stories; enforce Copilot on 5–13 SP stories.

Efficiency (Ticket Size)	Medium 7%, Large 7%, Small 5%	Small tickets underperform; medium/large better ROI.	Provide prompt recipes for medium/large; skip trivial small tasks.
Time Reduction Mix	Significant 7%, Moderate 12%, Slight 40%, None 8%, Not used 32%	Most tickets only “slight/moderate”; significant rare.	Upgrade prompts/context to shift slight → moderate/significant.
Usage by Size × Category	Small: 35% not applicable; Medium: balanced; Large: Improvement dominates (39%).	Copilot used well on large work; small tickets underleveraged.	For repetitive small tickets → one-click prompt templates.

🎯 Targeted Actions (Efficiency Lift)

Action	Expected Lift
Scale winners (Gen & Improve): Publish 6–8 prompt recipes (API handler+tests, refactor, migrations).	+0.4–0.8 pp avg time saved across all issues if “slight” → “moderate/significant” for 10%.
Decompose 30SP stories into 5–13 SP	Converts 0% → 8–11% gains.
Raise floor for small tickets: “gen + tests” micro-prompts.	Moves small from 5% → 6–7%; frees focus for bigger tasks.

Fix Quality category inconsistency: lint/fix prompts, static analysis, PR review checklists.	Converting 10% “no reduction” → slight adds measurable gains.
Shrink “not used” pool (32%): Target applicable subset (bugs, refactors, new endpoints).	Converting 10% not used → moderate adds +0.48 pp avg time saved.
