Engineering Time Audit

Most engineering teams assume they're spending most of their time building for customers, but they rarely measure it. This playbook provides a fast and practical way to audit where your engineering hours are truly allocated: features, infrastructure, compliance, fire drills, or duplicated work. No tools required, just a willingness to look under the hood and fix what's slowing you down.

Where Do Your Hours Go?

Why This Matters

You pay engineers to create value for your customers. Every hour diverted to plumbing, duplicate tooling, or after-hours firefighting chips away at that goal. This playbook demonstrates **how to measure the split, regardless of whether you have perfect data.** When you can quantify the drag, you can remove it.

Input Checklist & Quick Estimators

Input	Why It's Critical	Fast Way to Get a Number
Total Engineering-Hours / Month	Baseline for every other metric.	Head-count × 160 productive hours (40 h × 4 weeks) if you don't track time.
Feature Development %	Direct fuel for customer-visible change.	Filter last month's tickets/PRs tagged feature. If tags are missing, assume 50 % of engineer time for younger SaaS teams and 35 % for mature products.
Infrastructure Build & Automation %	Shows how much effort is spent on wiring/managing AWS instead of shipping features.	Count tickets/commits to IaC repos. No data? Start with 10% of Total Eng. Hours

Input	Why It's Critical	Fast Way to Get a Number
Technical Debt Reduction %	Investment that prevents tomorrow's drag.	Tickets labeled refactor, cleanup, upgrade. Lacking labels? Default to 10 % of Total Eng. Hours.
Unplanned Fire-Fighting %	Pure delivery drag—pages, reversions, manual fixes.	PagerDuty export → sum "Time to resolve". If no tool is available, multiply on-call hours by the average number of incidents per month.
Pattern Spread Score • Build paths (CI configs) • Runtime stacks (languages) • Infrastructure definitions (IaC vs scripts) • System Interface variants	Measures the cost of "too many ways to do the same job." More variants = more context switching.	See the next section for more info
Shared Systems Count • Infrastructure modules • CI/CD pipelines • Internal libraries/utilities • Network/topology configs • Monitoring/alerting baselines	Measures how much your organization avoids reinvention. The more teams share proven components, the more engineering time is allocated to product work instead of rework.	Inventory anything used across teams without modification: CDK/Terraform modules, reusable GitHub Actions, internal Python/Node packages, shared VPCs, or templated dashboards.

Pattern Spread Score (PSS)

Caveat: This part requires some finesse.

PSS measures architectural sprawl—how many different ways your team builds, runs, and connects code. It's not about "bad" patterns; it's about how much cognitive overhead engineers absorb just to get things done. Reducing variation and increasing reuse are key ways to reclaim engineering time.

TL;DR — Just want a fast score?

1. Count your patterns across four areas: **Delivery Methods**, **Code Patterns**, **Infra Types**, and **Interfaces.**
2. Plug them into this formula: $min\left(\sqrt{Delivery * Code * Infra * Interfaces}, 10\right)$
3. Use the result to find your penalty on the Engineering Scorecard.

Most small to mid-sized teams fall between **3 and 7**. Capped at **10** to avoid distortion.

What goes into PSS?

Category	Ask This Question	Count Examples
Delivery Methods	*How many distinct ways does your code get deployed or released?*	Monolith, microservices, Lambda, on-prem, mobile
Code Patterns	*How many languages, frameworks, or runtimes are actively used?*	Python/Django, Node/Express, Go, Java/Spring
Infra Types	*How many infrastructure platforms require configuration and maintenance?*	ECS, Lambda, Kubernetes, EC2, Kafka, Redis, Snowflake
Interfaces	*How does your code communicate across boundaries?*	REST, GraphQL, gRPC, event buses, file-based, CLI

Tip: Group minor variants (e.g., 3 REST APIs = 1), but count fundamentally different systems separately.

Example

Pattern Category	Count
Delivery Methods	2 (monolith + serverless)
Code Patterns	3 (Python/Django, Node, Go)
Infra Types	2 (ECS + Lambda)
Interfaces	2 (REST + gRPC)
PSS	$\sqrt{(2\times3\times2\times2)} = \sqrt{24} \approx 4.9$

PSS Penalty - (used below): $\min(0.05 \times \max(PSS-5,0), 0.3)$. Subtract 5% of feature hrs times leverage for every PSS above 5. Cap at 30%. Used in the Customer-Visible % metric below.

Turning Inputs into Metrics

1. Feature Hours= Feature Development Hours.

2. Infrastructure Tax Hours= (Infrastructure Build & Automation %) x Total Engineering Hours

3. Technical Debt Investment Hours= Tech Debt Reduction % x Total Engineering Hours

4. Fire-Fighting Hours= Unplanned % x Total Engineering-Hours

5. Pattern Spread Score (PSS)= √(delivery-path variants × runtime variants × infra config variants × system interface variants) , capped at 10

6. Leverage Multiplier= $1 + .2 * \log_2(\text{Shared Systems Count} + 1)$.

a. Sub-linear growth and cap the multiplier at 2.0 to maintain model realism.

7. Customer-Visible % = (Feature Hours × Leverage Multiplier – PSS penalty hours) ÷ Total Engineering-Hours.

Reading the Scorecard

Customer-Visible %	What It Tells You	Immediate Move
>65%	Engineers focus on value; plumbing is lightweight.	Fortify automated tests so quality keeps pace.
45–65%	Balanced, but efficiency gains are on the table.	Raise the Leverage Multiplier ≥0.8 by sharing CDK modules.
<45%	Delivery is stuck in the mud.	Consolidate duplicate build paths, migrate to AWS managed services, cut PSS in half.

An Example: 10 engineers 1500 hours/month

Here's a simplified example to show how the playbook works in practice. A 10-person team inputs their rough time estimates and receives a clear breakdown of where engineering hours are being allocated—and how much is translating into product output.

Input	Value
Feature Dev %	40%
Infra %	12%
Tech Debt %	10%
Fire-Fighting	8%
PSS Components	2× delivery × 3× runtimes × 2× infra × 2× interfaces → √24 ≈ 4.9
Shared Systems	5

Metrics

Metric	Result
Infra Tax Hours	.12 x 1,500 = 180
Tech Debt Hours	.1 ÷ 1,500 = 150
Fire-Fighting Hours	.08 x 1,500 = 120
PSS Penalty	(PSS = 4.9 → no penalty)
Leverage Multiplier	$1 + .2 * \log_2(5+1) \approx 1.52$
Customer-Visible %	(600 × 1.52 x (1 - PSS)) ÷ 1,500 = 61%

Interpretation: A well-balanced team with solid output and good tech-debt investment. PSS is under control, and the infrastructure cost is reasonable.

Now What? Turn Hours Into Dollars

Once you have the metrics, you can translate engineering time into dollar impact—whether it's surfacing waste, sizing technical debt risk, or demonstrating the payoff of standardization.

3 Ways Teams Leak or Reclaim Engineering Budget

Scenario	What to Watch	$ Formula
1. Overspending on internal work (Infra + Tech Debt + Fire-Fighting > 35%)	*Non-Customer-Visible Hours*	Waste = (NCV Hours – Target %) × Loaded Hourly Cost
2. Underfunding tech health (Tech Debt < 10% or Fire-Fighting > 10%)	Tech-Debt Hours or Fire-Fighting Hours	Future Drag = Deficit Hours × 4 × Loaded Hourly Cost
3. Pattern sprawl penalty (high PSS)	*PSS Penalty Hours*	Chaos Cost = PSS Penalty Hours × Loaded Hourly Cost

Rule of thumb: Every 10% misallocation in a 10-person org = $150k/year in hidden cost or untapped output.

The Virtuous Cycle: Standardization ROI

Making one investment to reduce sprawl can unlock hundreds of hours a month—here's how that plays out:

1. Standardize once → 2 platform engineers spend 4 weeks creating a hardened VPC + baseline pipeline.

2. Reuse across teams → 4 product teams adopt it; PSS drops 4 points and frees 80 hours/month/team.

3. Reinvest time → Feature hours go up, Customer-Visible % crosses 60%.

4. Compounding gains → Lower infra tax + higher reuse = faster delivery over time.

ROI Snapshot

Line Item	Value
Cost (2 eng × 4 weeks @ $10k/month)	$20,000
Hours saved in month 1 (4 teams × 80 h)	320 h
Value per hour (loaded)	$75
Payback	**< 1 month**

This is why tracking time and reuse matters—it turns small platform investments into exponential gains in product value.

Summing Up

The point of all this isn't to track hours for the sake of reporting—it's to reclaim control over your engineering investment. When you understand where the time goes, you can shift it. Shared systems beat one-offs. Simplicity beats cleverness. And standardization—done well—doesn't slow teams down, it frees them up. Use this playbook and **accompanying calculator** to drive sharper conversations and keep engineers focused on what moves the product forward.