Teams' Self Assesment

🧩 Heureka Tech Health Framework

Engineering Self-Assessment v1.0

🧭 Purpose & Instructions

The **Tech Health Self-Assessment** gives every team a consistent way to evaluate their technical health across **five key areas**.

- 📅 Run **once per quarter**.
- 👥 Led by **Tech Lead / EM / SE** with 1–2 engineers.
- 🎯 Goal: **reflection & improvement**, not judgment or ranking.
- 🗂️ Scale: **1 = chaotic / ad-hoc**, **4 = optimized & automated**.
- ✍️ Add short comments (“why this score”, “next step”).

🧩 How to Evaluate the Self-Assessment Results

Teams typically operate a mix of services — some **critical**, directly impacting customers or revenue, and others **supporting** or **shared** (internal tools, data pipelines, platform components). To keep the assessment both practical and meaningful:

- **Critical services** are evaluated **individually**, since their reliability and speed have a direct business impact.
- **Non-critical services** are **grouped into 1–2 clusters** representing similar characteristics (e.g., internal jobs, admin tools, or shared libraries).
- Each cluster is filled out as a single assessment, focusing on the *most representative* service within it.

This hybrid model avoids two common extremes — being too vague (one score per team) or too granular (one form per microservice). It ensures results stay **comparable, interpretable, and actionable**.

⚙️ Scoring Levels (applies to all axes)

Level	Description	Observable Example

1 – Chaotic	Work happens manually and unpredictably. Success depends on individuals, not systems. No clear ownership or repeatable process.	“Only one person knows how to deploy.” “Tests fail randomly.” “We rely on luck, not monitoring.”
2 – Emerging	Some structure exists, but inconsistently. Basic hygiene (reviews, tests, docs) appears, yet breaks under pressure.	“We have some tests, but not in CI.” “Manual review process; no clear release policy.” “Postmortems happen only after major incidents.”
3 – Defined	Core flows are standardized and repeatable. Ownership clear, metrics visible, incidents handled predictably.	“CI/CD reliable and repeatable.” “Alerts trigger known playbooks.” “ADRs document key architectural choices.”
4 – Optimized	System runs on data, not heroics. Quality and stability are measured, automated, and continuously improved.	“Pipelines validate changes automatically (e.g. checks quality gates and rules, SonarQube).” “Incidents auto-tracked and reviewed.” “Capacity balanced by SLO and error budgets.”

🧱 A. TECH DEBT (5 axes)

Sub-Axis	Level 1	Level 2	Level 3	Level 4
Code Quality Debt	No consistent reviews, duplication everywhere. Code inconsistent, high complexity, no shared standards or ownership.	PR reviews exist but no static checks. Manual reviews done inconsistentl y; no linting or static analysis; quality depends on individuals.	Linting & SonarQube in CI; code-owner rules. Consistent PR reviews; automated style and quality checks in CI; ownership defined.	Automated checks + refactoring backlog; clean metrics trend down. Static analysis, code smells tracked; refactoring backlog prioritized; technical debt trend improving (Sonar metrics down).
Architecture / Domain Debt	Monolith / tight coupling. Code and domains are tangled; any change affects multiple areas. No clear ownership.	Some domains defined; unclear API contracts. Partial separation, but boundaries fuzzy; APIs leak internal logic; unclear data ownership.	Clear bounded contexts; service boundaries reviewed. Domains explicitly defined; APIs stable and versioned; cross-domain dependencie	Architecture validated by Fitness Functions; ADR history maintained. Teams own independent bounded contexts; minimal coupling; contracts tested automatically

			s reviewed regularly.	and documented.
Infrastructure Debt	Fully manual process, inconsistent environments.	Some automation exists, but still requires manual steps or supervision	Reliable CI/CD with rollback.	Full infra-as-code; zero-touch deploy; blue-green or canary releases.
Process / Delivery Debt	Deploys happen anytime, nobody tracks what went out.	Some rules exist (code review, tagging), but people often skip them.	Releases follow a defined flow; everyone knows what's deployed and when.	Deploys fully automated; system tracks DORA metrics automatically.
Knowledge Debt	Context in people's heads.	README outdated; no ADRs.	Docs & ADRs for major services.	Continuous documentation; onboarding < 1 day.

🧪 B. TESTING & AUTOMATION

Sub-Axis	Level 1	Level 2	Level 3	Level 4
Unit Testing Coverage	No meaningful tests; manual validation only	Basic unit tests on key logic; not enforced in CI.	Solid coverage of critical code; tests run automatically in CI.	High-confidence unit suite; fast, stable, and used as living spec.
Integration Testing	No automated	Few manual or partial API tests.	Contract tests between	Full API contract validation in

	integration testing.		main services; checked in CI.	pipeline; auto-generated mocks and alerts on contract breaks.
E2E Flow Coverage	Only manual smoke tests after deploy.	Some automated E2E tests on main flows; run nightly.	Key business flows automated and part of CI.	Critical journeys fully automated; regression E2E suite runs on every deploy.
Pipeline Reliability	Builds often fail or give false results.	Pipeline sometimes flaky; unclear root cause.	Reliable CI/CD; success > 95 %, flaky < 5 %.	Self-healing pipelines; > 98 % success; issues visible and auto-reported.
Deployment Automation	Deploys done by hand (SSH, scripts, manual config). No rollback, high risk of error.	Basic scripts or CI jobs exist, but require manual input or approvals. Rollback unclear or manual.	One-click deploy from CI/CD; rollback tested and documented. No direct server access needed.	Fully automated delivery pipeline with health checks, canary or blue-green releases, and automatic rollback.

🔍 C. OBSERVABILITY & STABILITY

Sub-Axis	Level 1	Level 2	Level 3	Level 4
Monitoring Coverage	No metrics or alerts; rely on user reports.	Basic infra metrics (CPU/RAM); missing ownership.	Key services monitored with clear owners; SLOs for core SLIs.	Full observability stack (SLI/SLO/Error Budget) across services; live dashboards used daily by teams.
Alert Hygiene & On-Call Discipline	Alerts noisy or ignored; no on-call setup.	Some alerts tuned; on-call rotation informal or inconsistent.	Alerts actionable (>80%); OpsGenie used for ack/close tracking; P1/P2 severity respected.	On-call fully operational; all alerts actionable; MTTA/MTTR auto-tracked; no ignored pages.
Incident Response (MTTR)	Incidents ad-hoc; no tracking or ownership.	Incidents logged manually; response unstructured.	MTTR measured; Severity levels defined; RCA (Root Cause Analyses)required for P1s.	MTTR <1h; incident auto-created in OpsGenie; RCA auto-linked; lessons shared org-wide.
Logging / Tracing	Scattered logs; no correlation.	Structured logs; tracing partial on key flows.	Centralized logs; tracing active on top paths;	Full distributed tracing (100%); correlation

			searchable incidents.	IDs from edge to DB; logs auto-correlated with incidents.
Postmortems & Learning	None or blame-based.	Written ad-hoc; no follow-up.	Formal RCAs for major incidents; improvement items tracked.	RCAs for all SEVerity 1–2s; learnings feed Tech Health backlog; trends reviewed in Tech Health Review.

⚡ D. DELIVERY PERFORMANCE (DORA)

from Industry Benchmark from DORA reports

Metric	Level 1	Level 2	Level 3	Level 4
Deployment Frequency (DF) How often software changes are successfully released to production	< 1 deploy / month.	Weekly deploys.	Daily or per feature branch.	On-demand / continuous delivery.
Cycle Time (CT) Time from work item	10+ days	5-10 days	2-5 days	< 2 days

start → merged & deployed (includes review & QA).				
Lead Time for Changes (LTC) The time from code commit to deployment in production.	7+ days	2–7 days.	1–2 days.	< 24 h from commit to prod.
Change Failure Rate (CFR) The percentage of deployments that cause production incidents, rollbacks, or hotfixes.	25+ %.	15–25 %.	5–15 %.	< 5 %.
Mean Time to Recovery (MTTR) How long it takes to restore service after an incident	6+ hrs	2–6 h.	1–2 h.	< 1 h.

or degradation.				

🧠 E. GOVERNANCE & KNOWLEDGE

Sub-Axis	Level 1	Level 2	Level 3	Level 4
ADR Discipline	No record of decisions; context lost.	A few ADRs written reactively, not shared.	ADRs created for all significant technical or architectural changes.	ADRs standardized via template; reviewed quarterly; both team-level and global ADRs maintained.
Decision Traceability	No linkage to delivery work (Jira, Epics).	Partial linkage; some ADRs referenced manually.	Decisions traceable to delivery items in ≥80 % of cases where applicable.	Full traceability: each Epic or initiative links to its ADR; org-wide ADRs referenced in standards and playbooks.
Architecture Docs	None or outdated diagrams.	Partial or inconsistent C4 documentation.	Up-to-date diagrams for core domains; shared in team space.	Comprehensive Arc42-style documentation; automated generation from code/config.

Ownership Clarity	Undefined; “who owns this?” is unclear.	Some ownership lists exist, not maintained.	Clear ownership in Backstage; on-call mapping exists.	Ownership isn’t just written down — it’s proven by operations. The team that has it on the dashboard and gets the alerts owns it.
Transparency	Context closed within teams; decisions siloed.	Shared internally but not visible org-wide.	Visible across engineering; ADRs and docs accessible to all.	“Public by default” culture: decisions and lessons reviewed quarterly in Tech Health Review.

💬 Pulse Survey (Soft Signals – Monthly, 0-10 Scale)

Question	Purpose	**Example Interpretation**
1️⃣ How much does tech debt slow your team down?	Pain caused by accumulated/combined tech debt.	**Low (0–4):** “We avoid touching some modules because they’re too fragile.” → Plan refactoring or tech debt reduction. **High (8–10):** “We can

		deliver features fast; codebase feels clean."
2️⃣ How smooth and predictable is your release process?	Confidence in CI/CD and deployment flow.	**Low (0–4):** "Releases are stressful; builds fail or rollback often." → Review pipeline reliability. **High (8–10):** "Anyone can release safely; no surprises in production."
3️⃣ Do you have enough time for stability and engineering improvements?	Reality check for 25 % tech investment allocation.	**Low (0–3):** "We spend 100 % on BRs; stability work always postponed." → Leadership must rebalance capacity. **High (8–10):** "We consistently dedicate ~25 % to improvement and it shows."
4️⃣ Do you feel leadership supports technical investments?	Cultural alignment between product & engineering.	**Low (0–4):** "Every stability initiative gets deprioritized." → Requires CTO/CPO alignment. **High (8–10):** "Product managers understand why we invest in platform quality."
5️⃣ Are your tools and environments effective?	Developer experience and productivity blockers.	**Low (0–4):** "Local builds take 15 min; test envs unstable." → Identify and address DevEx issues.

		High (8–10): “Builds fast; environments reliable; good observability.”
➕ What’s currently the biggest thing slowing your team down?	Qualitative insight beyond metrics.	Examples: “CI queue delays”, “too many hotfixes”, “unclear ownership”, “missing test data”. Responses feed into Tech Health Review.

📈 Interpreting Scores

Average Score	Meaning	Action	**Leadership Focus**
1.0-1.2 ⚠️ Critical / Rewrite Candidate	Service or process is unsalvageable in its current form. Failures dominate. No automation, no ownership, unmaintainable code, or obsolete tech stack.	Stop new features. Stabilize to “safe to touch” state only. Prepare rewrite ADR with scope, dependencies, and migration path.	Escalate to CTO / Tech Health Review. Trigger rewrite / replatform decision. Secure capacity (via 25 % allocation or dedicated Big Rock).
1.2 – 1.9 🔴 Reactive / Unstable	Frequent incidents, manual steps, weak observability. Success	Launch stabilization sprint or Tech Recovery Plan. Identify 2–3 biggest blockers (e.g. infra, tests,	Assign SE+ support, increase stability capacity. Track CFR / MTTR weekly until ≥ 2.0.

	depends on heroic effort.	ownership). Document owner and flow.	
2.0 – 2.9 🟠 Emerging Discipline	Some structure works, but process breaks under pressure. Improvements ad-hoc, not systematic.	Introduce repeatable routines (release policy, postmortems, ADR template). Start automating manual tasks.	Keep teams focused on consistency. Avoid adding new scope until hygiene improves.
3.0 – 3.4 🟢 Stable Baseline	Delivery predictable, main risks visible, data collected but not yet fully automated.	Keep metrics trending up. Eliminate weak sub-axes (e.g. flaky tests, missing docs). Prepare next-level automation.	Protect 25 % engineering capacity. Reinforce that stability is paying off.
3.5 – 4.0 🔵 Optimized & Predictable	Fully automated flow, incidents rare and well-managed, improvements continuous.	Share playbooks, mentor other teams. Measure ROI of improvements (MTTR ↓ CFR ↓).	Use as internal benchmark and enablement partner; keep autonomy high.

🔄 Example Scenarios

- 🧱 **Architecture Debt = Level 2** → legacy monolith; plan for domain decomposition in Q2.
- 🧪 **Testing = Level 3** → solid CI; next step: flaky-test dashboard.
- 🔍 **Observability = Level 1** → no SLOs; urgent stability investment.
- ⚡ **Delivery = Level 4** → fast CI/CD; monitor CFR trend to keep risk low.
- 🧠 **Governance = Level 2** → ADR discipline inconsistent; add template & quarterly check.

🧩 Outcome

Teams own their improvement roadmap,
leadership sees comparable, trendable data,
and **tech health becomes a conversation, not a surprise**.
