ML and AI Technical Debt

BECM33MLE — Machine Learning Engineering

Dr. Jan Brabec[1]

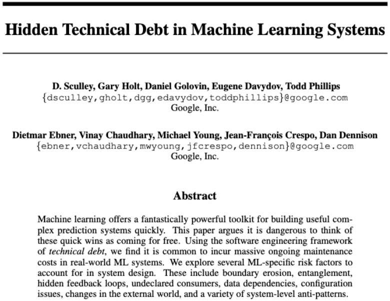
[1] CISCO

Motivation

- ML systems are complex socio-technical constructs combining data, code, and people. (see Lecture 4)
- Over time, they accumulate “hidden” costs that slow progress and reduce reliability.
- ML systems are **easy to prototype but difficult to maintain** at scale.
- **Goal:** Understand technical debt, cognitive load, and how to manage them in ML and AI systems.

Lecture Roadmap

1. Technical debt and cognitive load in traditional software
2. Technical debt in ML Systems (based on Sculley et al., 2015)
3. Managing and preventing debt
4. New frontiers: Technical debt in modern AI / LLM systems



Hidden Technical Debt in Machine Learning Systems

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips
`{dsculley,gholt,dgg,edavydov,toddphillips}@google.com`
Google, Inc.

Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison
`{ebner,vchaudhary,mwyoung,jfcrespo,dennison}@google.com`
Google, Inc.

Abstract

Machine learning offers a fantastically powerful toolkit for building useful complex prediction systems quickly. This paper argues it is dangerous to think of these quick wins as coming for free. Using the software engineering framework of *technical debt*, we find it is common to incur massive ongoing maintenance costs in real-world ML systems. We explore several ML-specific risk factors to account for in system design. These include boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, configuration issues, changes in the external world, and a variety of system-level anti-patterns.

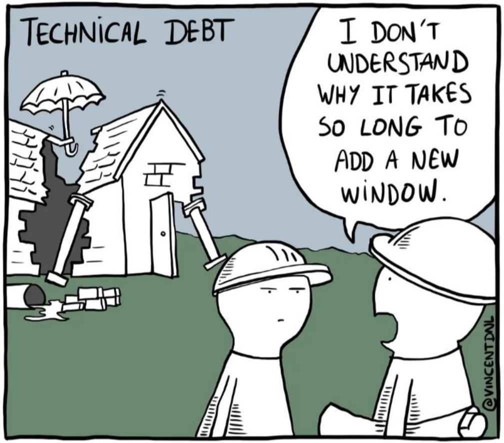
`https://proceedings.neurips.cc/paper_files/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf`

What is Technical Debt?

- Term introduced by Ward Cunningham (1992)
- Shortcut today ⟹ future cost tomorrow
- “Interest” = extra effort to modify or extend the system later
- Some debt is strategic, but unmanaged debt compounds.

> *“As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.”*
>
> — Meir Manny Lehman (1996)

What is Technical Debt?


TECHNICAL DEBT
I DON'T UNDERSTAND WHY IT TAKES SO LONG TO ADD A NEW WINDOW.
@VINCENTDNL

@vincentdnl

Types of Technical Debt

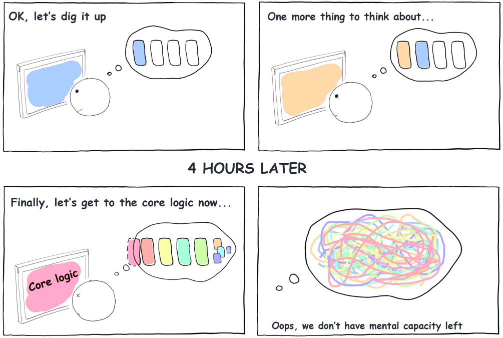
- Code debt (duplication, poor abstractions)
- Architecture debt (tight coupling, monoliths)
- Test and infra debt (lack of automation)
- Documentation debt (missing knowledge – ADRs (Why over how), . . .)
- **Cognitive consequences:** higher complexity, harder reasoning, easier mistakes

Cognitive Load and Technical Debt

- **Cognitive load:** mental effort required to understand, reason about, or modify a system.
- Technical debt increases cognitive load by obscuring structure and intent.
- Types of cognitive load:
  - **Intrinsic:** inherent system complexity (unavoidable)
  - **Extrinsic:** unnecessary complexity from poor design or process
- Goal: minimize extrinsic load to make reasoning easier.

> *"Cognitive load is how much a developer needs to think in order to complete a task."*
>
> — Zakirullin (2025), Cognitive Load Developer's Handbook
>
> `https://github.com/zakirullin/cognitive-load`

Cognitive Load


OK, let's dig it up
One more thing to think about...
4 HOURS LATER
Finally, let's get to the core logic now...
Core logic
Oops, we don't have mental capacity left

(c) Artem Zakirullin 2025 LICENSE: CC BY-NC-ND 4.0 x.com/zakirullin

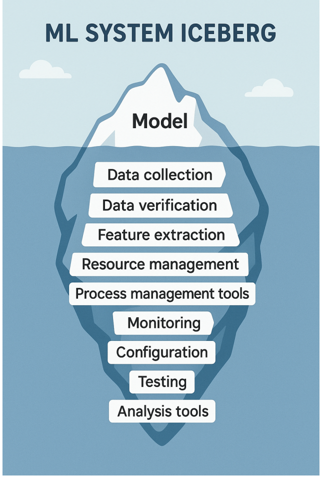
Software Engineering Technical Debt

- Technical debt in traditional software engineering has been studied for over 30 years.
- Research and practice mainly focused on code, design patterns, and architectural refactoring.
- These aspects are largely static — code can be analyzed, tested, and versioned.

From Pure Software to ML Systems

- Traditional software systems are **deterministic** — logic is explicit, outputs are predictable.
- Machine Learning systems are **data-dependent and probabilistic**.
- Many dependencies are **invisible** — hidden in data, configurations, and feedback loops.
- Small unseen changes can have large, system-wide effects.

The ML System Iceberg



Change Anything, Change Everything (CACE)

- In software, modular design allows local change with local effect.
- In ML systems, changing one component — feature, hyperparam, threshold — can have **global impact**.
- Example:
  - Feature stops being updated; downstream model silently misbehaves.
  - A feature's distribution drifts slightly; retraining causes different model paths.
- Data dependencies are harder to track and test than code dependencies.

Modular design requires clear boundaries — but where should they be?

Common boundary ambiguities in ML systems:

- Feature computation: ML system or upstream data pipeline?
- Data quality: Data platform responsibility or ML team responsibility?
- Model monitoring: ML engineers or infrastructure/ops team?

Example: Training-Serving Skew
Feature engineering implemented differently in training vs. serving:

- Training: Spark, counts purchases in last 30 days
- Serving: Pandas microservice, counts last 15 days (*bug!*)

Result: Model sees different features at serving time than training time

Boundary erosion: ML systems can "erode abstraction boundaries" through data dependencies, creating hidden coupling between components.

Correction Cascades

- **Pattern:** Model A (or set of rules and heuristics) learns to correct errors from Model B.
- When Model B is updated or fixed, A's compensating behavior becomes harmful.
- Each fix triggers a chain of compensating changes across dependent models.
- Over time, systems become fragile and brittle.
- **Improvement deadlock:** Improvement to a part makes the whole worse.

Epsilon Features and Feature Bloat

- Low-value “epsilon” features add long-term maintenance cost.
- Each feature increases retraining, testing, and dependency overhead.
- Remove obsolete or redundant features regularly.
- Simplicity often valued more than marginal accuracy gains.

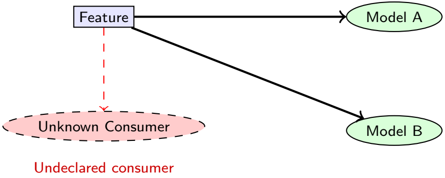
Epsilon Features and Feature Bloat (cont.)

- **Legacy Features:** Feature F included early, later made redundant by new features but removal goes undetected.
- **Bundled Features:** Group of features added together under deadline pressure, possibly including low/no-value features.
- ϵ-**Features:** Features adding minimal accuracy gain but high complexity overhead — tempting but costly.
- **Correlated Features:** Two strongly correlated features where one is causal, the other spurious. Model may credit both equally or pick the wrong one.
  - Risk: Brittleness if correlations change in production.

Common pattern: Features accumulate faster than they're removed.

Undeclared Consumers (Hidden Dependencies)

- **Definition:** A component or downstream system uses a model, feature, or dataset without declaring that dependency.
- Hidden consumers make safe changes nearly impossible:
  - You can't version or test what you don't know exists.
  - One "harmless" schema or model update can break a live system.
- Prevention:
  - Enforce explicit data and model contracts.
  - Use lineage tracking and registries.

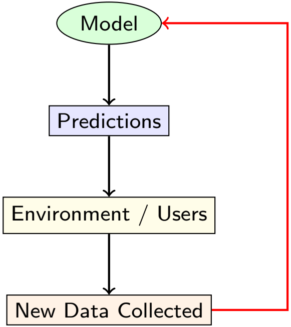


Feature and Model Reuse without Ownership

- Teams often reuse existing features or models without clear ownership.
- Leads to hidden coupling and difficulty in evolution.
- Example: shared customer embedding used in five pipelines—no one knows who maintains it.
- Prevention:
  - Feature registries with metadata and owner.
  - Versioning and deprecation policies.

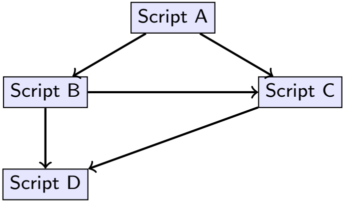
Hidden Feedback Loops

- **Definition:** Model predictions influence the future data the model is trained on.
- Creates a reinforcing cycle: predictions → environment → new data → retraining.
- Over time, this can:
  - Reinforce biases or amplify errors.
  - Mask underlying data drift.
  - Lead to unstable model behavior ("runaway feedback").
- Examples:
  - Recommender systems narrowing diversity.
  - Fraud detection reducing observed fraud → less signal.



Glue Code and Pipeline Jungles

- **Pattern:** ML systems evolve as ad-hoc scripts and connectors between tools.
- "Glue code" fills gaps between libraries, data sources, and teams.
- Over time: fragile "pipeline jungles" emerge.
- **Consequences:**
  - Untraceable data flow.
  - Slow, risky debugging and deployment.
  - High cognitive load for new team members.
- **Prevention:** Shared libraries, declarative pipelines (Airflow, Kubeflow), clear ownership.



Dead Experimental Code Paths

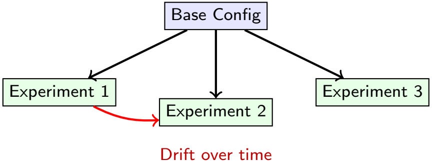
- ML codebases often accumulate unused models, experiments, and feature variants.
- It is often attractive to perform experiments as conditional branches in production code.
- These increase maintenance and testing cost without value.
- Prevention:
    - Periodic cleanup and refactoring.
    - Track experiment lineage and retire obsolete artifacts.
- Remember: "Every line of code is a liability unless it delivers value."

Configuration Debt in ML Systems

- **Definition:** Debt arising when system behavior depends on many loosely managed configuration parameters.
- In ML systems, configuration spans:
  - Data paths and preprocessing flags
  - Feature selections and model variants
  - Hyperparameters and thresholds
  - Deployment-time feature switches
- These settings are often undocumented and unvalidated.
- Configuration becomes code but without testing or review.

Hyperparameter Explosion and Copy–Paste Drift

- Hyperparameter tuning and feature experimentation multiply config files.
- Teams often copy working configurations to new experiments. Including unnecessary parameters.
- Over time:
  - Duplicated configs diverge silently (“drift”)
  - Unsure which config values are important and which just blindly copied
  - Behavior differs across environments
- Each new config increases maintenance and testing effort.



Customer-Specific Configuration Debt

- Business reality: different customers need tailored configurations (thresholds, feature flags, model variants).
- Each customer-specific config fork multiplies maintenance and testing burden.
- Without discipline, configs spread across codebase and diverge silently.
- **Problems:**
  - Hard to test all customer combinations.
  - Updates require cascading changes.
  - Exponential growth in QA effort.
- **Mitigation:**
  - Centralize configs in a single registry (e.g., feature store, config service).
  - Version and validate all configs with schema enforcement.
  - Abstract common patterns; isolate customer-specific overrides.
  - Automated testing for critical config combinations.

Dealing with Changes in the External World

- ML systems operate in **non-stationary environments**.
- Data distribution shifts due to user behavior, seasonality, or new sensors.
- Debt accumulates when systems assume the world is static.
- Example: ad click model decays as ad formats or user interests evolve.

Drift Detection

- **Prediction Bias:** predicted label distribution should roughly match observed labels.
- Track divergence over time — large shifts indicate drift or feedback issues.
- Slice metrics across key dimensions (region, segment, model version).
- Useful for early detection of world changes.
- **Action Limits:** enforce sanity bounds on system actions (bid caps, risky actions require confirmation, . . .).
- If limits are exceeded, trigger automated alerts or rollback.

Data Testing Debt

- If data replaces code, then data must be tested like code.
- **Data tests:**
  - Schema and range validation.
  - Statistical distribution checks (drift, missingness, outliers).
  - Temporal consistency and completeness.
- Data validation prevents silent failures and model corruption.
- **Consumer-driven contract tests:** downstream systems specify data expectations.

Reproducibility Debt

- ML experiments should be reproducible.
- Challenges:
  - Non-determinism from parallelization and random seeds.
  - Changing dependencies and environments.
  - Missing record of data versions and configs.
- **Mitigation:**
  - Version data, models, and configs together.
  - Log random seeds and runtime environment details.
  - Experiment management tools (MLflow, Weights & Biases).

From Hidden Debt to MLOps Practices

- Since 2015, tooling evolved: MLOps formalizes repeatable patterns.
- Explosion of tools: MLflow, Kubeflow, Airflow, DataBricks, SageMaker, . . .
- **Tools provide standardization and reduce glue code, but don't eliminate debt. They offer principled approaches for addressing it.**

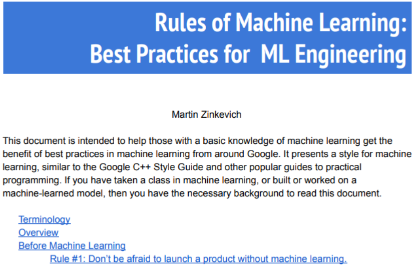
From Hidden Debt to MLOps Practices (cont.)

- Standardized workflows lower extrinsic cognitive load.
- Standardization makes onboarding easier.
- Features like e.g. *lineage tracking* available by default in MLOps platforms.
- Too many tools add orchestration debt: balance simplicity and automation.

Connecting the Dots: Rules o ML

- Throughout the past lectures we discussed challenges in ML system design:
  - data and ground truth management,
  - metrics and evaluation,
  - system design and technical debt.
- A classic reference that captures many of these lessons in concise, pragmatic form is:

  M. Zinkevich, "Rules of Machine Learning: Best Practices for ML Engineering," Google Research, 2017.
- Although written before modern MLOps, its guidance remains highly relevant:
  - start simple, automate, track, refactor.



Rules of Machine Learning:
Best Practices for ML Engineering

Martin Zinkevich

This document is intended to help those with a basic knowledge of machine learning get the benefit of best practices in machine learning from around Google. It presents a style for machine learning, similar to the Google C++ Style Guide and other popular guides to practical programming. If you have taken a class in machine learning, or built or worked on a machine-learned model, then you have the necessary background to read this document.

Terminology
Overview
Before Machine Learning
Rule #1: Don't be afraid to launch a product without machine learning.

`https://martin.zinkevich.org/rules_of_ml/rules_of_ml.pdf`

New Frontiers: AI System Debt

- Foundation and LLM-based systems introduce novel debt sources:
- **Prompt debt:** untracked, duplicated prompt templates.
- **Context debt:** uncontrolled tools and MCP dependencies.
- **Evaluation debt:** lack of automated, repeatable evals.
- **Model dependency debt:** reliance on external APIs.

Cognitive Load in AI Systems

- **Intrinsic load:** inherent complexity of stochastic, emergent model behavior.
- **Extrinsic load:** poor observability, complex prompts, weak abstractions.
- Example: reasoning about failures across prompt chains.
- Managing AI cognitive load: abstractions, monitoring, SWE/ML principles.

Conclusion

Thanks for your attention