Scaling ML Products At Startups: A Practitioner's Guide*

Atul Dhingra† Gaurav Sood‡

April 29, 2023

Abstract

How do you scale a machine learning product at a startup? In particular, how do you serve a greater volume, velocity, and variety of queries cost-effectively? We break down costs into variable costs—the cost of serving the model and keeping it performant—and fixed costs—the cost of developing and training new models. We propose a framework for conceptualizing these costs, breaking them into finer categories, and limn ways to reduce costs. Lastly, since in our experience, the most expensive fixed cost of a machine learning system is the cost of identifying the root causes of failures and driving continuous improvement, we present a way to conceptualize the issues and share our methodology for the same.

Keywords: scaling, startups, machine learning

1 Introduction

As an ML startup grows, so does the volume, velocity, and variety of data it generates and uses. Handling and managing this data efficiently and effectively is critical to the success of the startup. But both premature optimization in handling this data and leaving the job till too late can mean death. To guide thinking about which investment is needed and when, we have found it useful to divide ML startups into three stages. The first stage is the POC stage. At the POC stage, the cost is largely immaterial and the variety is moot beyond what is necessary for the POC. The second stage comports with the ability to handle modest volume, velocity, and variety, e.g., an automated contract ingestion company going from extracting data from one kind of contract to ten different contracts. In the second stage, the company needs to show progress in handling the volume, variety, and velocity of data while managing the burn rate. From the product side, the second stage can look like 10 POCs—the expected pace of delivery and the idiosyncratic nature of early customers conspire to limit serious investments in the

*The article benefited from comments by Pronoy Chopra, Noah Finberg, and Brian Whetter.

†USA, dhingra.atul92@gmail.com

‡USA; gsood07@gmail.com.

generalization of the model. The third stage is marked by the necessity to handle massive volume, velocity, and variety of queries, e.g., an automated contract ingestion company going from 10 to 1000 kinds of contracts. In the third stage, both the speed of scaling and the cost-effectiveness of scaling are major bottlenecks to a positive return on investment. Thus, substantial investments in infrastructure are generally required to enable the company to scale quickly and provide the service cost-effectively.

2 Cost

The cost of a machine learning model is the sum of the costs of building (and improving) the model, maintaining the model, and serving the model. Conventionally, fixed costs, e.g., the cost of engineering the system, are excluded from consideration when scaling a digital product. Instead, it is common to exclusively focus on the contribution margin. The theory goes that once you have a positive contribution margin, profitability is a matter of acquiring and retaining enough customers. However appealing the theory sounds, it is not without its chinks. Fixed costs are important because they affect the length of the runway. Without a long enough runway, very few big things will take off.

2.1 Variable Costs

We define variable costs as the total cost of serving the models and maintaining the models' performance. Conventionally, companies track variable costs via metrics like variable cost per customer, variable cost per dollar earned, or percentage of Annual Recurring Revenue (ARR). Monitoring the unit cost of production provides a natural way to look for the expected trend of declining marginal costs of production. To monitor specific cost centers, it is conventional to split total cost into different cost centers, e.g., storage costs per dollar of revenue, compute costs per dollar of revenue, etc.

Given the cost structure and cost rationalization opportunities, we split variable costs into three broad buckets:

1. **Compute costs.** Compute costs for serving large models can be substantial, especially at scale. However, any strategy aimed at reducing compute costs should price in the impact on business, which may stem from longer inference times, lower accuracy, etc. (Dhingra 2017). To better enumerate potential cost-saving opportunities, we divide the compute costs into the following pieces:

    (a) **The volume of queries.** All else being equal, the fewer the queries, the lower the prediction costs. To highlight potential opportunities to reduce the number of queries, consider the following example. Say that you are running a startup that uses machine learning to measure the speaker's emotions in movies. Most movies are shot at 24 frames per second. And we could set up the inference pipeline such that we predict emotions in each frame. But depending on the use case, a lower sampling rate may be enough.

(b) **The velocity of queries.** Serving infrastructure needs to be built with peak demand in mind. Even with adaptive scaling, the scale-up costs can be substantial. There are at least three broad solutions. The first is forecasting. Often, if you buy computing capacity ahead of time, it is cheaper. The second method is to constrain peak throughput. For instance, limiting the number of people who can shop at one time at a store as initial Amazon Go rollouts did (though plausibly for other reasons like controlling occlusion, theft, etc.). The last method is tiered pricing. For instance, ChatGPT offers a Pro tier that guarantees access during peak demand.

(c) **Model size.** By model size, we mean the memory and the number of computations needed to infer. Memory Access Cost (MAC)(Ma et al. 2018) and Floating Point Operations (FLOPs) are typically used to measure these costs and are often part of the model's KPIs. The price of a larger model is generally greater latency and greater memory and compute requirements. The benefit is greater accuracy, resolution, e.g., higher resolution of the generated image, etc.

- **Memory.** The GPU memory required to load an ML model to make predictions on the target deployment platform can be affected by numerical precision, e.g., fp32, fp16, int8, the number of parameters in a model, etc. Memory requirements affect cost for two reasons. Memory costs money. For instance, with a fixed number of FLOPs, a GPU with larger memory costs more. Second, memory can be seen as a constraint on deployment options. For instance, if the model cannot fit an on-premise device, a GPU, or a VM, you have to go with the cloud, a pricier GPU, and a pricier VM respectively.
- **Compute.** The total number of MAC or FLOPs needed for a prediction. It is a function of the model capacity (the depth and the width of the network).

We can reduce the model's memory and compute footprint using techniques like Knowledge Distillation (Hinton, Vinyals and Dean 2015), Quantization, Graph surgery, TensorRT, etc. But it is useful to not assume the impact of these methods on latency. We have encountered cases where model distillation reduced the number of parameters by 500x but still increased latency because of the sequence and the nature of the calls.

2. **Bandwidth costs.** The cost of shuttling data for audio-visual applications can be substantial. Shuttling data generally means greater latency and lower reliability. The standard ways of reducing bandwidth costs include compression and downsampling. For instance, the resolution needed for accurate prediction may be lower than the resolution at which the image or video is captured.

3. **Operations Cost.** Operations costs pool over the cost of people and tools needed to keep the model performant. For instance, for annotation in a self-driving car company, the costs include the cost of hiring a driver, the labeling team, and the annotation tool (and its associated contract or development costs). These costs can be amortized between the cost of building new models or improving models and keeping the model performant. To

reduce the operations burden, we need to optimize data collection and annotation. To improve the speed of annotations features like predictive typing, pre-annotations, etc., can be useful. To improve the returns on the data collected, it is useful to build a rough ROI function of the type common in active learning. For instance, in some cases, we may want only driver miles during peak hours or when the weather is bad. Other strategies include using synthetic data to drive performance gains (Sagers et al. 2022).

2.1.1 Cloud Vs. On-Premise and Rent Vs. Buy

Compute and bandwidth costs, the cost of updating the model over the air, the cost of latency, and the cost of building and maintaining your own infrastructure, e.g., issuing security patches, monitoring and predicting hardware failures, are some of the considerations in deciding between cloud and on-premise deployment. The key virtue of cloud is low startup costs, which is important in the era of high developer costs. It also keeps the focus on innovating in the value-generation step. Even if the cloud is the expensive option over the long term, using it initially fits with the oft-repeated "build first, optimize later" mantra.

3 Fixed Costs

Fixed costs cover the costs of building and improving models. The costs can be broken down into hardware costs, e.g., cars in an AV company, LIDARs, etc., and development costs, which can be further broken down into the cost of developers and the cost of model training infrastructure. Hardware costs can be amortized between variable and fixed costs based on how much of the infrastructure is used for maintaining performance and how much is used for building new models.

3.1 Hardware Costs

Sensor costs include cars, LIDARs, cameras, microphones, lighting, scanners, RFID tags, etc. It is conventional to amortize hardware expenditures based on longevity and expected sale value when they are swapped out. Maintenance and breakage costs are generally baked into hardware costs and listed under service and breakage costs.

3.2 Development Costs

Development costs are a major part of fixed costs. There are two big cost centers—the cost of the model training infrastructure and the cost of developers.

3.2.1 Training Infrastructure Costs

The cost of model training infrastructure includes the cost of storage, the cost of hardware for training the models, and the cost of other engineering tools. These costs can balloon very

quickly and need to be carefully modeled and projected to figure out which costs need attention. Infrastructure can also be a bottleneck. While metrics for development velocity are less easily derived, they are probably the most important metrics. We have used self-reports through surveys to quantify the effect of better infrastructure on development speed and found that investments in data comprehension and infrastructure for data analytics for root-causing problems are generally worth it. With that, we have some specific advice about what not to do.

The five most common infrastructure-related mistakes that startups make are:

- **Choosing the wrong tool for the job.** For instance, for storing user attributes for Fortnite, where not all attributes are available to all users, using a NoSQL solution is better than RDS. The root cause could be a bias towards building with what you are familiar with and unawareness of reference architectures. The solution is educating yourself about the reference architecture and the use case.

- **Reinventing tools.** Given the developer costs, inventing developer tools for ML startups should be an option of last resort till well into the third stage or unless the math really tallies in favor of such an investment. The thumb rule is that no developer problem is novel. If your company is struggling with a problem, it is likely that other companies are too. And major cloud service providers or another company generally have some answer to that problem. Add to it the point that rarely is the ROI on the perfect solution substantially greater than a good enough solution. For instance, tools like SageMaker can be an effective way to deploy models until the company is well into the third stage. Lastly, when picking cloud tools, another maxim is: when in doubt, go serverless.

- **Not carefully modeling how costs will increase with scale.** The growth in costs in infrastructure is often non-linear. Before marginal costs come down, they often increase. If you do not carefully model the costs, the rise in costs will likely come as a shock, leaving little time for crafting an effective mitigation strategy.

- **Ignorance of cost optimization opportunities.** Simple principles of infrastructure savings—volume discounting, e.g., using forecasting to buy compute in bulk, auto-scaling, using resources during times when they are cheaper, e.g., training costs can be brought down by 80—90% by using spot instances but require checkpointing, exist across the board.

- **False Optimization.** Companies sometimes just move the pain than save money. For instance, some companies do not enable logs because of concerns about storage expenses. (These concerns can be overcome by setting up ways to archive and eventually delete the logs.) And the consequence is that it is much harder (and takes much longer) to diagnose errors.

3.2.2 Developer costs

Developer costs can be split into three kinds of costs:

- **The cost of engineers.** All else being equal, the number of engineers required to serve customers at a particular quality grows sharply with the diversity, volume, and velocity of the data. The only way to reduce the slope of the function is by increasing the maturity of the infrastructure and paying down technical debt. Till the second stage, reducing the slope may not be optimal but in the third stage, it becomes sine-qua-non.

  The cost of engineering churn. The cost is a sum of two costs:

  - **The cost of losing tribal knowledge.** At startups, many critical pieces of how the system works reside within people and good documentation is generally unavailable. Losing an engineer means losing a lot of the context which makes development much slower and also means that engineers have to relearn the same lessons which can be expensive (and time-consuming). Given that startups cannot often afford redundancy—multiple engineers responsible for the same piece of code—in the extreme, an engineer leaving can have dramatic consequences for the maintenance of a particular feature.
  - **Hiring costs.** Hiring is expensive and time-consuming. It costs the remaining engineers interview time. It means generally that the company has access to one less engineer so the delivery speed is slower than optimal. It also includes the cost of ramping up another engineer. Add to all of this the fact that, if all else is equal, losing an engineer increases the odds of another engineer leaving.

  There are many reasons for regrettable engineering churn. Aside from compensation, and the conditions in the wider job market, some engineers leave because of poor quality of life. Some common aspects of startup life—the amount of work, frequent reorganizations, and frequent movement across workstreams—make working at a startup more unpleasant. Beyond that, as the company moves to the third stage, engineers can be increasingly stuck fixing bugs if the company has not paid down technical debt. And it is a canon among engineers that fixing bugs is not as satisfying as building new features. Engineers also generally think it is particularly unappealing to fix other people's bugs and 'simple' bugs that come with a large overhead. Because of these reasons, for many engineers, the more time spent fixing bugs, the less pleasant the job. The time devoted to fixing bugs is some function of the quality of the code and infrastructure, and the quality of tools used to root cause errors. For all of these reasons, paying down technical debt should be moved forward. We will discuss how to think about technical debt in the next section.

- **The cost of lower product quality.** The cost of a lower-quality product is customer churn, poor reputation, and lower sales. One measure of product quality is the number of customer complaints (multiplied by their impact—often crudely coded into simple categories, e.g., high, medium, and low—and the time they remain unsolved) that stem from bugs. This metric aligns quality in ways important to the customer. It is also conventional to add uptime as a KPI.

Nearly all software ships with bugs and often the bugs are known—the bugs are there because of agreed-upon trade-offs, which often go undocumented. So when using data on bugs to decide whether or not to invest in operational excellence, it is useful to focus on unanticipated bugs.

In our experience, the number of unanticipated bugs is weakly related to conventional code quality metrics, e.g., the extent of logging, the number and coverage of tests, the quality of documentation, etc. The largest predictor is the quality of engineers and oversight.

- **The cost of lower development velocity.** The consequences of a lower cadence of feature delivery and longer times to deploy to a new site or customer are stark—poorer sales and greater cash burn (and the associated risk of running out of money). The conventional metrics for tracking this problem are
  - **The frequency of releases.** (Or the rate of feature delivery, pro-rated for complexity). Many big engineering problems eventually cause the release cadence to become slower. The rate of production needs to be broken down into its own input metrics. For instance, are machine learning engineers spending most of their time wrangling data? Is a lack of data versioning what is causing the release process to fail?
  - **The time to release to a new customer or site.** Over time, we expect this metric to decline sharply. This is a metric that needs to be broken down and understood for scaling.

3.2.3 Technical Debt

Startups are financed by technical debt. It is generally imprudent to aim for zero technical debt. The level at which you engineer solutions should reflect the needs and constraints of the time. For instance, if you are testing your ETAs for a route planning algorithm in a second-stage company, you may write a script against Google Maps routing. Eventually, as the needs evolve, you may want to write a class or add the ability to query other mapping services. Similarly, at the start of some ML projects where the data generation process is expected not to change, it may not be useful to monitor performance against fresh samples as designing and maintaining such systems is expensive. Instead, like other engineering solutions, e.g., regular-expression-based systems, it may be reasonable to rely on customer complaints to flag problems. On the flip side, it pays to take on the right kind of debt and pay down high-interest debt. So how do we think about which technical debt should be paid down first? The core principles are:

- **Pay down the most expensive debt first.** The cost can be customer value or the cost of maintenance.
- **Solving upstream technical debt is generally more important than downstream technical debt.** In ML, paying down data quality issues is better than building solutions to deal with data quality in the model. For instance, it is generally more cost effective to

fix sensors or their integration than to spend time on making machine learning models more robust to sensor malfunction (though eventually those investments are needed).

Not all technical debt is a result of engineering to the needs or constraints of the time. Much of it stems from bad engineering practices. Debt avoided is money saved. Following the meta engineering principles for writing maintainable software, software that is easily operable, e.g., easy to troubleshoot and run, simple, e.g., which can be achieved through abstraction, etc., can save a lot of money.

4 Maintaining Performance

Maintaining performance over time is a three-step process: 1. defining and monitoring performance, 2. root-causing failures, and 3. finding and deploying the fixes. The last part is too problem specific so we limit our discussion to the first two points.

4.1 Defining and Tracking Performance

Maintaining performance starts with defining performance objectives that capture all relevant costs to the business. For instance, the performance of a credit card fraud detection algorithm can be defined using two metrics: the sum of the cost of fraud (a consequence of false negatives—transactions that were fraudulent but not flagged) and the cost of bad customer experience (a consequence of false positives—transactions where no fraud was committed but that were flagged). The costs need to be comprehensive. For instance, the cost of poor customer experience needs to take into account direct and indirect costs, e.g., the cost of resolving the issue, the impact on customer retention, and the impact on business reputation. The other Key Performance Indicators (KPIs) may include the latency of the model and the cost of serving a prediction.

When monitoring metrics, it is easy to be misled by small spikes. One way to avoid fixating on false positives is to smooth the time series using an exponentially weighted moving average or Kalman filter. Smoothing, however, comes at the cost of lower sensitivity. One way to increase sensitivity is to increase the sample size. Larger samples are also essential if you want sensitive measures for various slices of the data. Another reason for spikes is (expected) variation from seasonality and such. For instance, an ice cream manufacturer may adjust sales for seasonal variation. To the extent that variables like seasonality reflect factors outside the control (as in the example above rather than issues like poor posture recognition of people in winter clothes), it is useful to regress those out.

4.2 Dashboards

To triage performance, we need to regress out known things so that we can focus on unknown unknowns. For that reason, dashboards should reflect all known sources of variation in the performance metric. Say that we know we perform worse on occluded images. We can account

for the issue by adjusting for the proportion of occluded images or by showing performance on occluded and unoccluded images and having a panel that tracks the proportion of occluded images over time. Another helpful way of laying out the dashboard is to lay it out as a funnel that roughly maps to the system diagram, with each output metric split into input metrics.

4.3 Identifying the Root Causes of Problems

There are three parts to problem-solving: 1. Coming up with a potential set of explanations, 2. Putting priors on the explanations, and 3. ruling out rival explanations. We can generate explanations by analyzing the funnel, e.g., where the dropoff is happening, sampling failures, and looking at correlations, e.g., performance is lower in a certain subgroup. To come up with priors, it is common to use prior experience or principles like proximal explanations are more likely. One way to prioritize the search is to go in order of probability though often it is useful to think about some of the probable causes and see if you can exclude some. One way to isolate causes is to look for which explanation explains all the data. The reason this strategy works is that there is often only a single point of failure. For other strategies, see Shroff and Sood (2023).

4.4 Model Troubleshooting

Traditional software engineering practice embraces strong abstraction boundaries using encapsulation and modular design to create maintainable code in which it is easy to make isolated changes and improvements. But "machine learning models are machines for creating entanglement and making the isolation of improvements effectively impossible." IML systems follow the "CACE principle: Changing Anything Changes Everything" (Sculley et al. 2015). This is partly why explainable AI that promises to explain why the model predicted X vs. Y is considered a promising tool for understanding problems (Duckworth et al. 2021). But we have a long way to go with explainable AI and in our ability to come up with good causal explanations. For that reason, it helps to have a conceptual typology of why models fail.

4.4.1 Typology of Conceptual Issues That Cause Models to Fail

Models fail for six broad conceptual reasons:

1. **Bad data.** By bad data, we mean cases where tooling or coding errors corrupt the data. For instance, an aggregation pipeline does not account for missing data on one of the feature vectors and produces Nans.

2. **Wrong y.** Cases where the data generation process for the dependent variable has changed such that the assumptions used to build the model no longer hold. Say, for instance, you bootstrap a recommendation model using click data from randomly picked news articles. Say that the model is used to generate the set of articles that the user sees and what the users click on then is used to further refine the model. In this case, the data generation

process is continuously changing as the last period of data affects what articles people see and click on. One of the common risks here is showing a very narrow category of articles to the user. This can cause a large deterioration in our ability to learn and optimize for user preferences, which may change over time.

3. **Missing features.** Features limit what can be learned. If say you are building a housing price predictor and do not have data on road noise that can be heard from the house, and say that it is an important predictor of housing prices and is weakly correlated with other features, then there is little chance of reducing the error stemming from the omitted variable.

4. **Limited representation.** Even when the variable is present, the representation of the feature (data) limits what relationships can be learned. For instance, in sentiment classification, the order of words matters. If you use a bag of words model, there is no way to learn about sequences.

5. **Extrapolation error.** The kind of data that the model is predicting on is different from what it was trained on. And that means the assumptions behind the model no longer hold.

6. **Interpolation error.** Interpolation error stems from two sources—(local) underfitting and (local) overfitting. There are multiple sources of interpolation errors. First, cross-validation, the primary tool we use to prevent overfitting is too crude to prevent local overfitting. Generally, model losses are too insensitive to local overfitting and underfitting across narrow slices of data. The second source of issues is model capacity. If you have a linear model and the true function is quadratic, it will lead to interpolation errors. One of the solutions to both local overfitting and local underfitting is more data. To fix underfitting to the data, it is also helpful to increase model capacity.

4.4.2 Detecting Model Problems

Like other logic built on the assumption that data distribution is stationary, e.g., Regex, machine learning applications can fail silently. Any change in the data-generating process needn't affect the business metrics or indeed even the model's topline metrics but it is useful to be aware of the shifts. Here are some ways to measure the consequences of changes in the data-generating process:

1. **Error on annotated data.** In supervised learning applications, one way to flag problems is to continuously annotate new data. To trigger an investigation, one heuristic is that performance on the latest batch is lower than previous batches. Usually, we only annotate a small sample in each time period, and sampling variability alone could cause metrics to fluctuate. To address that, we can smooth the estimates and adjust them for known-knowns. For instance, if you have a model that predicts the ETA of the truck, you may want to adjust for scenarios where the prediction is known to be worse, e.g., the proportion of truck loads scheduled to pick up a load during peak hours.

2. **Distribution of prediction confidence intervals.** Detect drift indirectly by nonparametrically testing the width of the confidence intervals.

3. **Out of range.** Monitor the entry of new classes (or in the case of continuous variables, whether the values are beyond the previously observed range) in the dependent and independent variables.

4. **Concept drift.** Is the composition of a class changing? The ways to detect it include monitoring the covariance, correlation, clusters, etc., in a class.

5. **Data drift.** In our experience, monitoring generic aspects of the covariates has not been particularly fruitful in detecting or root-causing errors. But it is good data hygiene. Methods like tracking the reconstruction error have proven more effective (Zavrtanik, Kristan and Skočaj 2021).

4.5 Tools

Different tools are needed for different scales. For instance, companies rarely invest in internal tools at the first stage. In the second stage, companies focus on tools that help you dive deep into the product vertical you are trying to capture, e.g., an AV company, which has completed a successful POC on a single state highway will likely need to invest in data exploration tools to solve L4 automation for all US highways. However, once companies surpass the second stage, they need a way to store, represent, and visualize the data in a more indexable manner. Typically, you need a feature store to not just capture variance within data, but also replicate model degradation on a large, growing set of input data.

5 Conclusion

Scaling startups is hard. Challenges unique to machine learning make scaling machine learning startups harder. In this paper, we provide a framework for thinking about costs and highlight some cost-saving opportunities. We also outline ways to think about one of the most expensive portions of machine learning—root causing problems.

References

Dhingra, Atul. 2017. "Model complexity-accuracy trade-off for a convolutional neural network." *arXiv preprint arXiv:1705.03338* .

Duckworth, Christopher, Francis P. Chmiel, Dan K. Burns, Zlatko D. Zlatev, Neil M. White, Thomas W. V. Daniels, Michael Kiuber and Michael J. Boniface. 2021. "Using explainable machine learning to characterise data drift and detect emergent health risks for emergency department admissions during COVID-19." *Scientific Reports* 11(1):23017.
URL: *https://doi.org/10.1038/s41598-021-02481-y*

Hinton, Geoffrey, Oriol Vinyals and Jeff Dean. 2015. "Distilling the knowledge in a neural network." *arXiv preprint arXiv:1503.02531* .

Ma, Ningning, Xiangyu Zhang, Hai-Tao Zheng and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. pp. 116–131.

Sagers, Luke W, James A Diao, Matthew Groh, Pranav Rajpurkar, Adewole S Adamson and Arjun K Manrai. 2022. "Improving dermatology classifiers across populations using images generated by large diffusion models." *arXiv preprint arXiv:2211.13352* .

Sculley, David, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo and Dan Dennison. 2015. "Hidden technical debt in machine learning systems." *Advances in neural information processing systems* 28.

Shroff, Sid and Gaurav Sood. 2023. "Problem Solving.".

Zavrtanik, Vitjan, Matej Kristan and Danijel Skočaj. 2021. "Reconstruction by inpainting for visual anomaly detection." *Pattern Recognition* 112:107706.