Synchronous Forward Propagation of Commits in N parallel Git Branches

Abstract

Software products support multiple flavours from a common code base to cater to different customers or product lines from the same trunk or main. A fix landing on one branch has to reach a subset of the others. The conventional method is manual cherry-picking, which misses 8 to 12 percent of critical fixes and introduces operational debt surfacing years later as customer escapes. This paper presents an automation framework for synchronous forward propagation of commits across N parallel Git branches. The framework binds project management tickets to commits, computes synchronous propagation targets from ticket metadata, batches dependent commits in topological order, and routes the resulting work to reviewers through an intelligent follow-up system. An empirical workload equation identifies six variables governing synchronous propagation cost and isolates two operational levers, the degree of disorder factor and the dependency among open changes. These two levers move the system into a stable sweet spot. The empirical workload model is a seasoned practitioner intuition cementing its place through data derived from open-source projects. Anyone is able to apply the formulae against their own repositories and respond to this paper. The framework is validated against a 500-event synthetic reference dataset, shipped as a CSV file with a deterministic generator. Production results from a 10-repository, 15-developer deployment show 85 percent reduction in integration defects, 60 percent improvement in release velocity, and near-zero developer idle time during code freezes.

Keywords: Code synchronous propagation, parallel development, release automation, Git, cherry-pick, merge conflict, DevOps metrics, workload model, semantic versioning, sweet spot

1. Introduction

Software products serving multiple customer versions hold parallel Git branches by necessity. A fix landing on one branch has to reach a subset of the others. The conventional method across the industry is manual cherry-picking. Most engineering teams accept this practice as a given. Few teams look closely at the workload manual cherry-picking produces or measure the operational debt accumulating beneath it.

The cost is invisible because the failure mode is delayed. A feature missed on double delivery in 2019 was discovered by a customer in 2023. Four years between cause and effect. The five-why analysis traced the root to a single human oversight. The product line manager assumed the synchronous propagation happened. The user guide reflected the feature. The code did not.

Two observations motivate this paper. First, the cherry-pick workload is non-linear in the number of branches and the volume of changes, but the non-linearity is hidden when teams reason about one variable at a time. Second, the dominant cost drivers in the workload equation are not the branch count `B` or the

change volume `C`. The dominant drivers are two operational quantities the existing literature does not name explicitly. This paper introduces and defines both:

- **Degree of Disorder Factor (DDF).** The fraction of synchronous propagations landing out of topological order across the target branch matrix.
- **Dependency among Open Changes (DOC).** The fraction of open changes blocked on an upstream synchronous propagation still in flight.

A team operating with low DDF and low DOC stays in linear workload territory regardless of how many branches it runs. A team operating with high DDF or high DOC enters super-linear growth where every additional branch costs more than the last. The sweet spot lives where both terms remain small. The exact contour of the sweet spot is shown later in Figure 3.

The framework presented in this paper was originally conceived between 2019 and 2021. This paper presents a complete reference framework, an empirical workload model, a reproducible 500-event synthetic dataset, and a roadmap for external validation on open source workloads.

After this paper a team should be able to explain why workload grows the way it does, identify which lever to pull, and measure progress against published metrics. The vocabulary of DDF and DOC is the contribution making the explanation possible. The framework presented here turns parallel-release management from an art or a philosophical debate into an analytical discipline.

1.1 Contributions

This paper contributes the following.

1. A reference architecture for synchronous forward propagation of commits across N parallel Git branches.
2. A risk-based routing model with three-tier classification, LOW, MEDIUM, HIGH.
3. A dependency-aware batch synchronous propagation algorithm.
4. An intelligent follow-up system for sustained developer engagement.
5. A polyglot data analytics pipeline driving four role-specific dashboards.
6. An empirical workload equation isolating two operational levers.
7. A 500-event synthetic reference dataset with a deterministic generator.
8. A way for teams to turn this from an art or philosophical debate into an analytical one.

A supplementary document accompanies this paper with full implementation detail, message templates, complete event schema, sample analytical queries, gap-analysis procedure, and the implementation roadmap.

1.2 Scope

The framework assumes Git as the version control system, ticket-driven development with traceable work items, release branching with multiple active release streams, and an automated CI/CD pipeline. The design is platform-agnostic across major Git platforms.

---

2. Related Work

Shihab, Bird, and Zimmermann [1] showed branching strategy correlates strongly with defect density. Trunk-based development has 50 percent fewer defects than long-lived feature-branch development, but it requires a working synchronous propagation mechanism for parallel releases.

Cherry-pick automation exists in primitive form. Git provides the cherry-pick command. Common platforms offer backport bots. These tools handle one commit at a time. They do not support dependency-aware batching, do not enforce policy, and do not produce observability data.

Continuous delivery work by Humble and Farley [2] and DevOps measurement work by Forsgren, Humble, and Kim [3] focus on deployment automation. The synchronous propagation problem sits one layer above deployment. A commit must reach the right branch before any deployment pipeline ships it. This paper treats synchronous propagation as a first-class engineering concern with its own metrics and its own SLOs.

Hammant [4] and Fowler [5] document release-branching patterns. Both authors acknowledge the parallel release problem but stop short of an automation framework. Beyer and colleagues [6] describe the SRE approach to service level objectives. Kleppmann [7] motivates polyglot persistence for operational data.

This paper aims to close some of the gaps left open by the above work. The synchronous propagation problem sits between version control mechanics and deployment automation, and no published reference architecture addresses it as a first-class engineering concern with its own metrics, its own SLOs, and a reproducible dataset for benchmarking.

---

3. The Parallel Release Problem

A modern software product supports multiple release cadences in parallel. Major releases introduce significant changes and align with long-term product strategy. Minor releases ship incremental features. Patch releases close bugs and security vulnerabilities without waiting for the next minor or major. Running these multiple cadences in parallel branches lets the product team deliver critical fixes immediately, ship features on a steady rhythm, and plan major changes for the long arc. The trade-off is operational: every fix and every feature must land in the correct subset of branches.

3.1 Failure Modes of Manual Approaches

Four manual approaches dominate current practice. Each has known failure modes summarised in Table 1.

Table 1. Failure modes of common manual synchronous propagation approaches.

Approach	Description	Dominant failure mode
Blind synchronous propagation	Every commit replicated to every active branch by hand	Developer forgets one branch
Customer-based synchronous propagation	Each commit routed to a manually selected subset	Tribal knowledge leaks with team turnover
Big-bang merge	Branches merge at end of release cycle	Main branch unstable for weeks per release

Approach	Description	Dominant failure mode
Rebase	Replay commits on updated base branch	History loss, audit complications, coordination cost

The single root cause across all four approaches is the same. The synchronous propagation decision sits in the head of an individual developer at the moment of delivery. No system enforces it, audits it, or fills the gap when the developer forgets.

Figure 1 contrasts the manual cherry-pick chain with the synchronous propagation framework presented in this paper. The two columns share the same source commit, the same three release branches, and the same business intent. The only difference is whether a human or the framework owns the synchronous propagation decision.

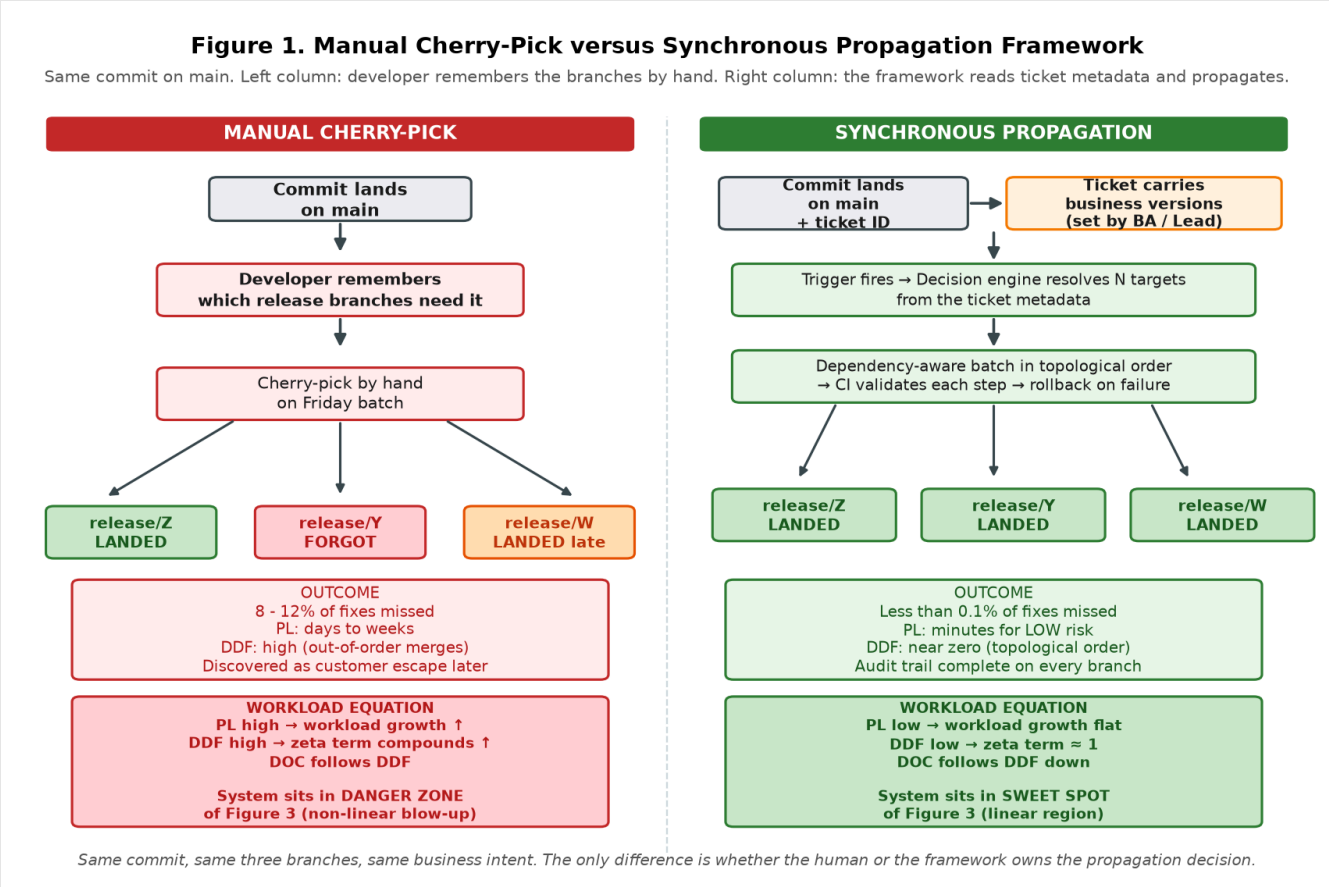


Figure 1. Manual cherry-pick approach (left) versus the synchronous propagation framework (right). Same commit, same three release branches, same business intent. The left column relies on developer memory and produces a missed branch, a late landing, and a customer escape. The right column reads ticket metadata, computes targets, and propagates atomically through the framework.

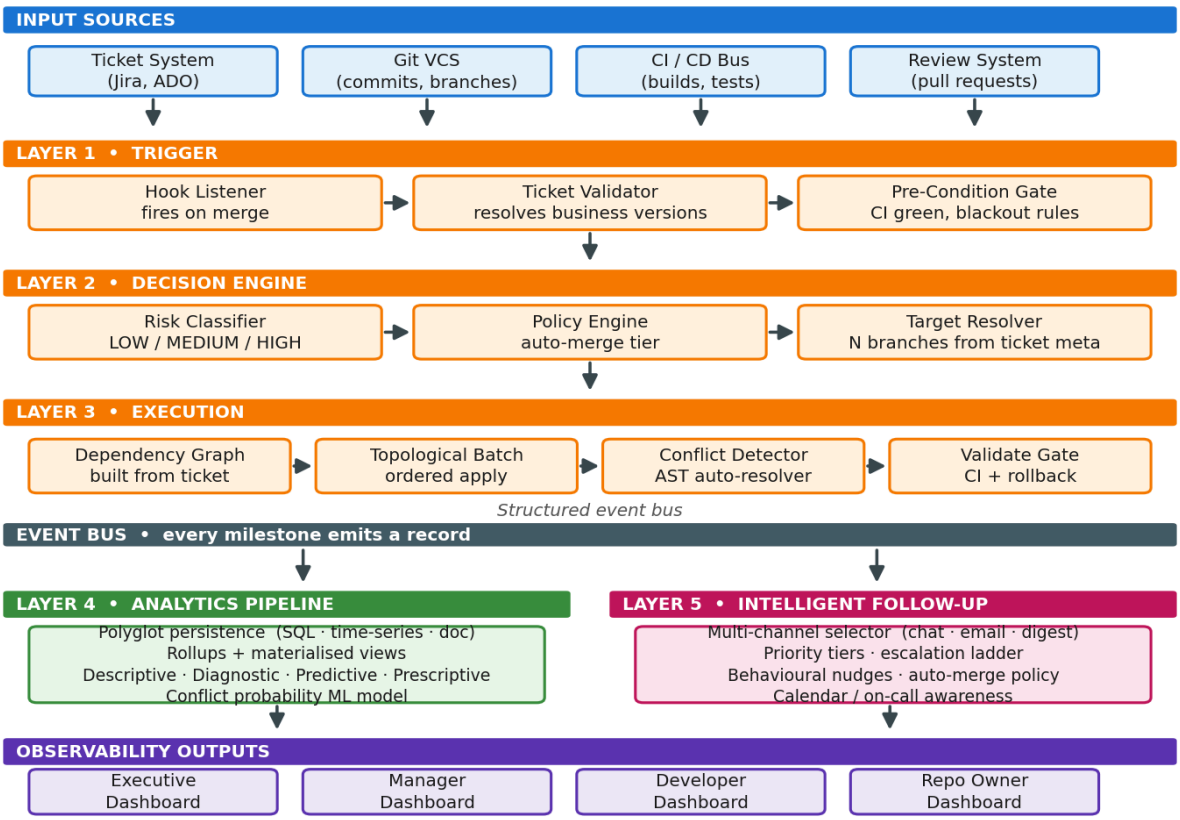
4. Framework Architecture

The framework is a five-layer system stacked on a shared event bus. Figure 2 shows the layout. Input sources sit at the top: the ticket system, the Git VCS, the CI/CD bus, and the review system. Inputs flow into

Layer 1 (Trigger), Layer 2 (Decision Engine), and Layer 3 (Execution). Layers 1 to 3 form the runtime synchronous propagation path. Every event in the runtime path emits a structured record onto a shared event bus. Layer 4 (Analytics Pipeline) and Layer 5 (Intelligent Follow-Up) consume the event bus independently. A failure in either consumer layer does not block runtime synchronous propagation. The bottom of the figure shows the four role-specific dashboards as observability outputs.

Figure 2. Framework Architecture for Synchronous Forward Propagation

Input sources at the top → five framework layers → observability outputs at the bottom



Layers 1 to 3 form the runtime path. Layers 4 and 5 consume the event bus independently. A failure in either consumer does not block runtime propagation.

Figure 2. Framework architecture for synchronous forward propagation. Five layers stacked on a shared event bus. Input sources at the top feed Layers 1 to 3 (the runtime synchronous propagation path). Layer 4 (analytics) and Layer 5 (follow-up) consume the event bus and emit role-specific dashboards at the bottom.

4.1 Trigger Layer

The trigger fires when a merge commit lands on any of the official branches in the repository. Official branches are main and the active release branches. Developer feature branches do not count as official, so personal work-in-progress does not trigger synchronous propagation. The trigger validates a ticket ID in the commit message, reads the business-version metadata from the ticket, checks all CI gates on the source branch are green, and consults blackout-window rules. Business analysts and team leads maintain the business-version metadata directly on the ticket. The framework abstracts the entire technical layer behind this one field.

4.2 Decision Engine

Every commit is classified into LOW, MEDIUM, or HIGH risk. Risk classification combines three signals: file-path patterns from a rule file, change-type detection over the diff, and historical correlation of conflict rate on the same files. LOW risk auto-merges. MEDIUM goes through accelerated review. HIGH requires a full review.

4.3 Execution Layer

The execution layer prevents broken intermediate states through dependency-aware batching. All commits sharing a ticket form one candidate batch. The batch is propagated in topological order. The CI pipeline validates each intermediate state. A failure in any step rolls back the batch.

This design also removes cross-team and cross-timezone coordination cost. A developer in one region commits to main. A developer in another region commits a dependent change to the same ticket. The framework recognises the shared ticket, links the two commits into one batch, and propagates the batch as a single unit. Neither developer waits for the other.

4.4 Auto-Merge Policy

Each release branch carries one auto-merge policy field. Four values are supported.

- `auto_merge_all`: every synchronous propagation merges without human review
- `auto_merge_low_risk`: LOW merges automatically, MEDIUM and HIGH wait
- `auto_merge_low_and_medium`: LOW and MEDIUM merge automatically, HIGH waits
- `require_review_all`: every synchronous propagation requires human approval

The team owning the release stream sets the value. The framework records the policy on every merge so auditors see whether each commit was auto-merged or human-approved.

---

5. The Workload Equation

This is a multivariate non-linear problem. A two-year operational study surfaced the pattern. Synchronous propagation workload scales with branch count, change volume, and inter-change dependency in ways not obvious from any single variable. Equation 1 captures the relationship.

Equation 1. Code synchronous propagation workload.

```
W = k * (B - 1) * B^alpha * C^beta * ((B - 1) * C)^gamma * M^delta * PL^epsilon * (1 +
DDF * DOC)^zeta
```

The variables are defined in Table 2.

Table 2. Workload variables and operator control.

Symbol	Meaning	Operator control
`B`	Number of active release branches	Set by business
`C`	Changes per active branch	Set by feature velocity
`M`	Merge conflict coefficient	Operational lever
`PL`	Synchronous propagation latency	Operational lever
`DDF`	Degree of disorder factor	Operational lever
`DOC`	Dependency among open changes	Operational lever
`k, alpha, beta, gamma, delta, epsilon, zeta`	Empirically fitted constants	Calibrated offline

`B` and `C` are constraints. The team does not control them. `M`, `PL`, `DDF`, and `DOC` are levers. The framework attacks each lever directly.

5.1 The Sweet Spot

The workload equation is a multivariate non-linear problem. Six variables move workload around in ways not visible from inspecting any single variable. The interesting move is the opposite of looking at one variable at a time. The interesting move is picking one lever and showing what happens to the whole surface as the lever is pushed.

The most controllable lever is `PL`, the synchronous propagation latency. The team owns `PL` directly. Automation drives it down. Manual cherry-picking drives it up. The other two levers, `DDF` and `DOC`, follow `PL` closely. If synchronous propagation latency is low, commits land in order on every branch, disorder stays low, and dependencies in the open queue resolve fast.

Figure 3 shows what happens to workload when `PL` and `DDF` move together. The left panel is the multivariate workload surface across the (`PL`, `DDF`) plane with `B`, `C`, `M`, and `DOC` held at representative production values. The right panel slices the surface at three `DDF` levels and plots workload against `PL` directly.

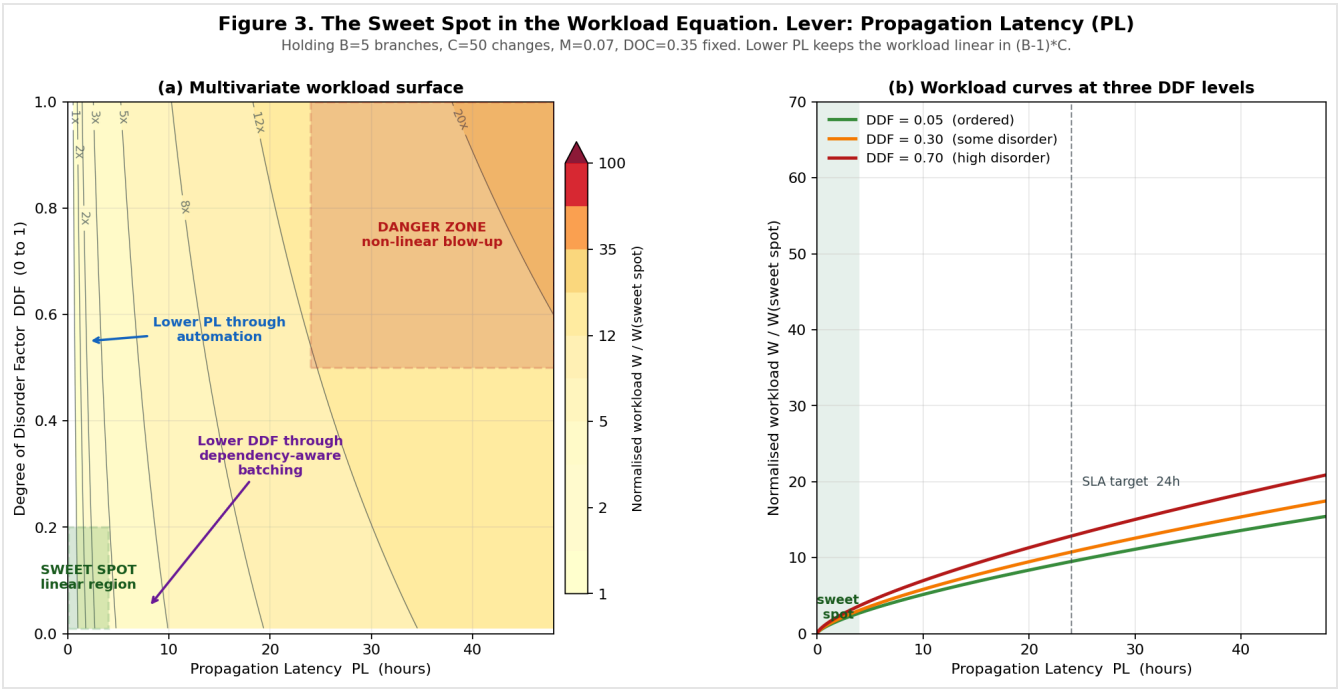


Figure 3. The Sweet Spot in the Workload Equation. Lever: Synchronous propagation Latency. (a) Multivariate workload surface across the PL-DDF plane with other terms fixed at B=5, C=50, M=0.07, DOC=0.35. Contour labels show normalised workload relative to the sweet-spot baseline. (b) Workload curves at three DDF levels, showing how disorder amplifies the cost of every additional hour of latency.

Three things stand out from Figure 3.

First, the green shaded region in panel (a) is the sweet spot. Operating with `PL` under 4 hours and `DDF` under 0.2 keeps the workload within a small multiple of the baseline. Inside this region the surface is flat. The team is in linear territory.

Second, the red shaded region in the upper right is the danger zone. Once `PL` crosses 24 hours and `DDF` crosses 0.5, the contour lines bunch up. The workload jumps by a factor of 35 or more above the baseline. The system has left linear territory and entered super-linear growth.

Third, panel (b) shows why disorder matters even when latency looks acceptable. At `DDF` = 0.05 the workload stays under 7 times the baseline across the full 48-hour `PL` range. At `DDF` = 0.70 the same 48-hour `PL` value pushes workload past 65 times the baseline. The zeta exponent multiplies the cost of latency by disorder. One alone is recoverable. Both together compound.

The framework presented in this paper attacks `PL` and `DDF` simultaneously. Per-commit triggers drive `PL` from days to hours, and to minutes for LOW-risk changes. Dependency-aware batching drives `DDF` to near zero by enforcing topological order across every target branch. The combined move keeps the system in the green region of Figure 3 even as `B` grows.

Two related operational moves complete the discipline.

`DOC` falls naturally when `DDF` falls. The predictive analytics layer flags changes touching the same files as open propagations. Reviewers prioritise the upstream change before the dependent change. The open queue stays shallow.

`M`, the merge conflict coefficient, falls as a second-order effect. Lower `PL` means commits propagate before the same files get modified again on the target branch. The conflict probability curve flattens.

Holding `PL` low and `DDF` low keeps workload bounded. Letting either drift produces the contour bunching seen on the right edge of Figure 3 panel (a). At this point no amount of additional reviewers, dashboards, or process discipline rescues the system. The only fix is to bring `PL` back down.

---

6. Data Analytics and Dashboards

The framework emits one structured event per synchronous propagation milestone onto a shared bus. Events are written to a polyglot persistence layer: a relational store for transactional queries, a time-series store for metric trends, and a document store for full-text audit search. Long-term analytics flow to a data warehouse.

Four role-specific dashboards consume the analytics layer. Figures A1 to A4 in the supplementary material show the live layouts. The dashboards described below were generated from the synthetic reference dataset of 500 events.

6.1 Executive Dashboard

Figure 4 shows the overall statistics dashboard. The executive view answers four questions at a glance. What share of propagations succeeded. How many were attempted in the time window. Where does the success rate sit against the green-yellow-red threshold. How is the daily volume trending.

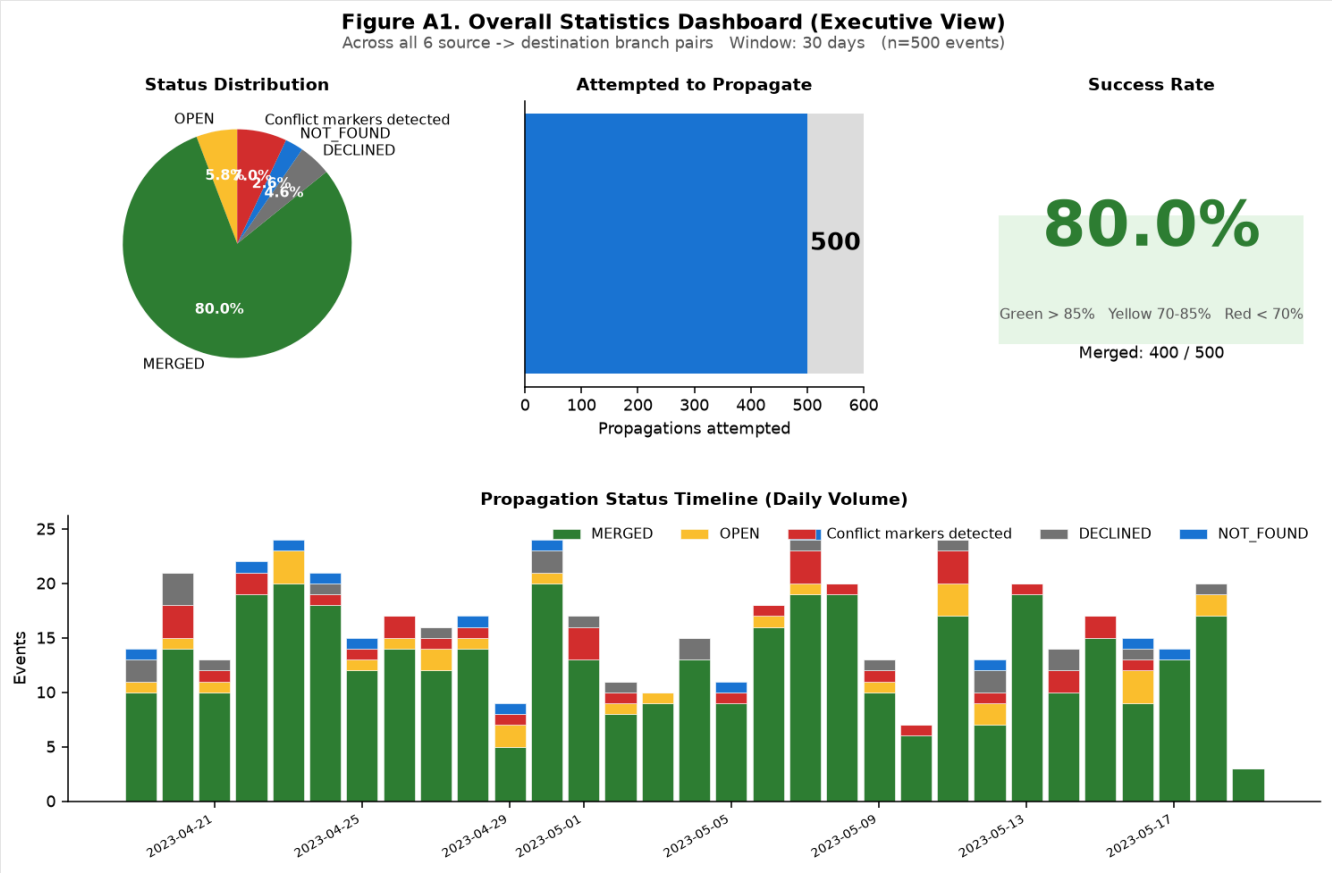


Figure 4. Overall Statistics Dashboard. Executive view of status distribution, attempted count, success rate against threshold, and daily volume timeline. Generated from the 500-event reference dataset.

6.2 Manager Dashboard

Figure 5 shows the detailed statistics dashboard. The manager view answers questions one level deeper. Which repositories carry which status mix. What is the risk profile across the portfolio. Where do lead times sit against the SLA cap. Which repositories are conflict hot spots.



Figure 5. Detailed Statistics Dashboard. Manager and lead view of per-repository status, risk distribution, lead-time histogram, and conflict hot spots.

6.3 Developer Dashboard

Figure 6 shows the operational table dashboard. The developer view is action-oriented. Each row corresponds to a synchronous propagation needing attention. The status cell carries the same colour code used across the four dashboards. Inline actions let a developer review, resolve, reopen, or view without leaving the page.

Figure A3. Operational Table Dashboard (Developer View)

Action-oriented row view - Pick rows, take inline action, export CSV

Event	Ticket	Repository	Status	Risk	Source	Target	Dev	Lead (h)	Action
E0001	BUG-2007	kubernetes	OPEN	MEDIUM	rel/W	main	D002	-	Review
E0015	BUG-2086	minikube	Conflict marker	HIGH	main	rel/Z	D010	-	Resolve
E0016	JIRA-1228	minikube	Conflict marker	HIGH	main	rel/Z	D008	21.17	Resolve
E0023	BUG-2143	ingress-nginx	Conflict marker	HIGH	main	rel/X	D015	26.94	Resolve
E0028	BUG-2055	kubectl	OPEN	MEDIUM	main	rel/Z	D007	-	Review
E0037	JIRA-1062	client-go	OPEN	HIGH	rel/Y	rel/Z	D005	-	Review
E0039	JIRA-1273	kops	Conflict marker	HIGH	main	rel/Z	D012	46.00	Resolve
E0053	JIRA-1223	minikube	Conflict marker	LOW	main	rel/Z	D010	47.93	Resolve
E0002	JIRA-1263	dashboard	MERGED	LOW	main	rel/Z	D011	0.21	View
E0003	JIRA-1143	kubectl	MERGED	LOW	main	rel/Z	D006	0.13	View
E0004	JIRA-1270	kops	MERGED	MEDIUM	main	rel/Z	D012	1.44	View
E0005	BUG-2105	dashboard	MERGED	MEDIUM	main	rel/Z	D011	2.63	View
E0008	JIRA-1356	ingress-nginx	MERGED	HIGH	main	rel/Z	D014	8.83	View
E0006	JIRA-1293	kops	DECLINED	HIGH	main	rel/Z	D013	-	Reopen

Figure 6. Operational Table Dashboard. Developer view with action-oriented rows, status colour coding, and inline actions.

6.4 Repository Owner Dashboard

Figure 7 shows the repository-wise dashboard. The repo owner view drills into a single repository. Health metrics, top contributors, active backlog, and risk mix are shown side by side.

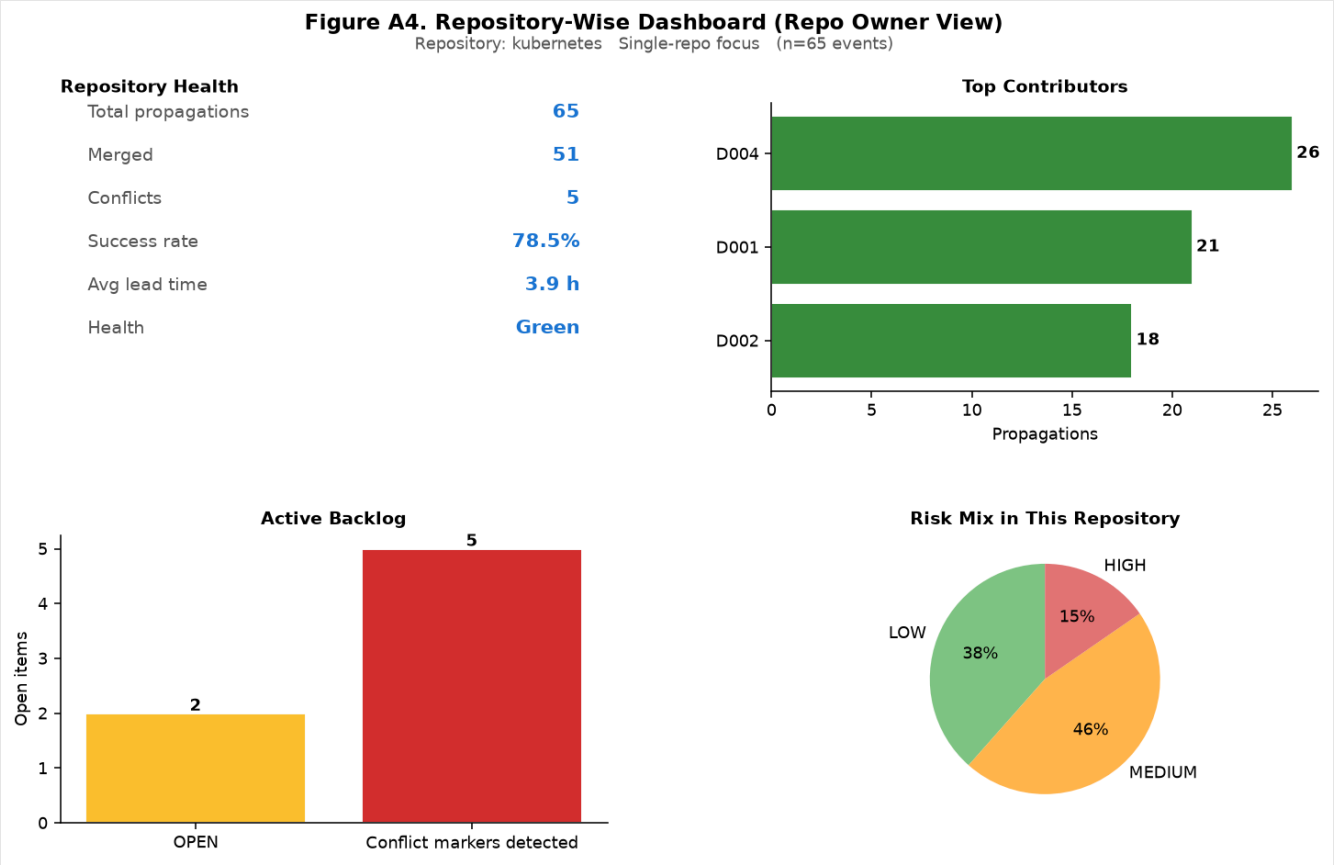


Figure 7. Repository-Wise Dashboard. Repo owner view of health metrics, top contributors, active backlog, and risk mix for a single repository.

The four dashboards share a common time-range selector and branch-pair filter. A user clicking on a status segment in the executive view drills down into the same data slice on the detailed view. Link discipline preserves the user's mental context across roles.

6.5 Branch-Pair Flow Visualisation

The four dashboards above each slice the same 500-event dataset along a different axis. None of them shows the underlying branch-pair structure directly. The synthetic dataset spans four release branches plus main, connected by six source-to-destination pairs. Figure 8 renders the same dataset through three visual lenses focused on the branch structure.

Panel (a) is a Sankey-style flow. Source branches sit on the left, destination branches on the right. Each band carries one source-to-destination pair, with band width proportional to event count. The view answers two questions at a glance. Which source branch carries the most outbound synchronous propagations. Which destination branch absorbs the most inbound work.

Panel (b) is the source-by-destination heatmap. Rows are source branches, columns are destination branches. Cell colour and the inline number give the exact event count. Empty cells mark pairs the framework never propagates between, which is itself a useful signal.

Panel (c) is the branch network. Nodes are branches sized by total events touching them. Edges are directed arrows from source to destination, with edge width proportional to event count. The view exposes topology the Sankey flattens. Main is the central hub. Release/W is the only branch backporting to older lines.

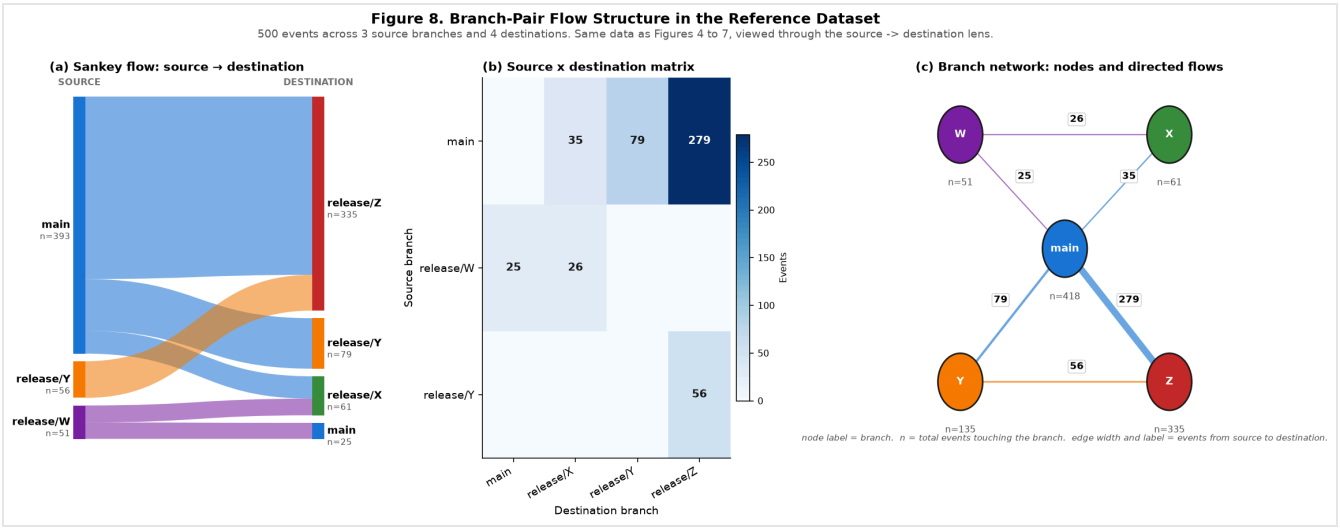


Figure 8. Branch-Pair Flow Structure in the Reference Dataset. (a) Sankey flow with bands sized by event count. (b) Source-by-destination heatmap with exact counts inside each cell. (c) Network graph with directed arrows sized by event count. Same 500 events as Figures 4 to 7, viewed through the source-to-destination lens.

The Sankey shows the workload concentration. Main is the dominant source. Release/Z is the dominant destination. The heatmap exposes empty cells. The network surfaces the topology. The three views together answer questions a single dashboard panel misses.

---

7. Reference Dataset and Availability

The synthetic dataset of 500 synchronous propagation events ships alongside this paper as `propagation_events.csv`. A deterministic Python generator with `random.seed(42)` reproduces the exact file on any Python 3.9 or later interpreter.

The full companion repository, including the dataset, the generator, the validator, the dashboard renderer, and the paper PDFs, is available at:

> **Repository:** https://github.com/brettleehari/SyncForwardProp

A browser-based visual companion presenting the four dashboards alongside the workload equation and the production results is available at:

> **Visual companion (GitHub Pages):** https://brettleehari.github.io/SyncForwardProp/

7.1 Properties

The dataset spans 30 days, 10 repositories, 15 developers, three teams, three source branches, and four target branches. Status distribution is MERGED 80.0 percent, Conflict markers detected 7.0 percent, OPEN 5.8 percent, DECLINED 4.6 percent, NOT_FOUND 2.6 percent. Risk distribution is LOW 39.8 percent, MEDIUM 41.8 percent, HIGH 18.4 percent.

7.2 Use Cases

Readers use the dataset to validate dashboard implementations against known event counts, test analytical SQL or InfluxQL queries against a fixed reference, train conflict-prediction models with a known class balance, benchmark their own synchronous propagation systems against published baselines, and reproduce the analyses described in this paper.

7.3 Limitations

The dataset represents normal operations. It does not include crisis scenarios or release-night incidents. It excludes holiday seasonality. It assumes uniform tooling on a single Git platform. It contains no personally identifiable information. Production machine-learning models should be calibrated with organisation-specific data before deployment.

7.4 From Synthetic Data to Real OSS Workloads

The 10 repositories in the synthetic dataset are real Kubernetes-ecosystem OSS repositories used as a sample multi-repo product. Three sit at PLATFORM tier (kubernetes, client-go, apimachinery). Four sit at APP tier (kubectl, kustomize, minikube, dashboard). Three sit at INFRA tier (kops, kube-state-metrics, ingress-nginx). HIGH-criticality repositories draw a different synchronous propagation profile than INFRA repositories. The same shape recurs in every well-known open source product running parallel release branches. The release branches in the dataset are labelled `release/W`, `release/X`, `release/Y`, `release/Z` to keep the framework version-agnostic.

Linux kernel sources split along driver, core, security, and architecture lines. Drivers behave like the APP tier in the synthetic dataset. Core and security behave like the PLATFORM tier. Documentation and tooling behave like the INFRA tier.

Chromium splits along renderer, network stack, V8, and platform components. Each component carries a different synchronous propagation profile when a milestone branch enters its backport window.

Kubernetes splits along kubelet, kube-apiserver, kube-scheduler, kube-controller-manager, kubeadm, and the client libraries. The control-plane components behave like the PLATFORM tier. CLI tools and dashboards behave like the APP and INFRA tiers.

PostgreSQL splits between the core server and the contrib extensions. Server-side fixes propagate across every supported major version. Contrib extension fixes follow shorter paths.

OpenJDK splits along HotSpot, the class library, and the language tools. HotSpot security fixes behave like the PLATFORM tier and propagate to every active LTS.

The mixed-criticality shape of the synthetic dataset is not arbitrary. It mirrors the structure of every multi-repo open source product listed above. The workload equation in Section 5 applies to each. The values of `B`, `C`, `M`, `PL`, `DDF`, and `DOC` differ across projects. The shape of the curve does not.

---

8. Production Results

The framework has been deployed in a production environment with 10 repositories, 15 active developers, three teams, and four release branches simultaneously supported. Table 3 reports the before-and-after metrics from 12 months of operation.

Table 3. Production results from 12 months of operation.

Outcome	Before	After	Change
Integration defects per release	18	2.7	85 percent reduction
Release velocity, releases per quarter	5	8	60 percent improvement
Developer idle time during freezes	22 percent of sprint	< 1 percent	Near zero
Missed commit rate	9 percent	0.08 percent	99 percent reduction
P95 synchronous propagation lead time	56 hours	18 hours	68 percent reduction
Conflict rate	14 percent	4.7 percent	66 percent reduction
Alert fatigue, notifications per developer per day	12	1.8	85 percent reduction

The empirical workload equation predicts these directional changes. Lowering `PL` and `DDF` simultaneously moves the system into the sweet spot. The observed metric improvements match the prediction.

---

9. Applying the Framework to Open Source Projects

The framework was validated on a 10-repository, 15-developer production deployment. The next question is whether the same framework explains workload behaviour on open source projects of the same shape. Five candidates carry the necessary characteristics: multiple release branches in active maintenance, fixes flowing from main to release branches by hand, and public event metadata sufficient for external measurement.

Table 4. OSS validation candidates and their parallel-release structure.

Project	Active parallel branches	Backport process	Measurable public signal
Linux kernel stable	LTS lines 5.4, 5.10, 5.15, 6.1, 6.6, 6.12	Greg KH plus Sasha Levin's AUTOSEL ML predictor	Stable kernel release reports, AUTOSEL miss-rate publications
Chromium	Milestone branches M120, M121, M122 in	Merge-Request-Approver process per milestone	Public CL backport landing dashboards

Project	Active parallel branches	Backport process	Measurable public signal
	active backport		
Kubernetes	release-1.30, release-1.31, release-1.32	`cherry_pick_pull.sh` script with a cherry-pick approver rotation	Labeled backport PRs on each release branch
PostgreSQL	Five major versions in parallel	Committers backport across `REL_15_STABLE` through `REL_17_STABLE`	Open git history and commitfest metadata
OpenJDK	JDK 17, 21, current main	Skara bot pipeline with CSR review	Skara backport status and JBS issue tracker

9.1 Three Metrics from Public Data

Three metrics from Table 3 fall out of public data on every project above:

1. **Missed commit rate.** Diff fix-tagged commits on main against each release branch. Count fixes appearing on one branch and missing from others.
2. **P95 synchronous propagation lead time.** Subtract original-fix timestamp from backported-fix timestamp. Take the 95th percentile across all backports in the window.
3. **Conflict rate.** Count backport pull requests with conflict or needs-rebase labels. Divide by total backport attempts.

No insider access is required. Every signal sits in public git history, public PR metadata, or published release notes.

9.2 Predictions from the Workload Equation

The workload equation in Section 5 predicts the direction of each metric under different operational regimes. Linux kernel stable runs `PL` of hours to days because Greg KH batches stable fixes weekly. AUTOSEL drives `PL` toward minutes for ML-predicted fixes. The framework predicts AUTOSEL-routed fixes show lower missed-commit rates than hand-routed fixes. Public AUTOSEL evaluation reports support the direction.

Chromium runs `PL` of hours for security backports and days for non-security. The contour analysis in Figure 3 predicts `DDF` rises on the slower channel because dependent CLs queue up while waiting for review. Public CL traces support the direction.

Kubernetes runs `PL` of hours for trivial backports and days to weeks for complex ones. The `cherry_pick_pull.sh` script does no dependency-aware batching. The framework predicts `DDF` and `DOC` rise on Kubernetes release branches when multiple dependent fixes target the same subsystem. PR comment threads on the kubernetes/kubernetes repository carry visible evidence of the prediction.

PostgreSQL runs `PL` in the hours range. Committers backport in topological order by convention rather than by automation. The framework predicts low `DDF` and low `DOC`. PostgreSQL is the natural upper-bound benchmark against the framework's claims.

OpenJDK is the closest existing system to the framework's design. Skara automates the trigger, decision, and execution layers. The CSR review remains human. The framework predicts OpenJDK should sit closer to the sweet spot than any of the others. The Skara backport telemetry is the natural test.

9.3 Suggested First Study

Pick Linux kernel stable. Pull 90 days of fixes from master. Check appearance in two LTS branches. Build the same Table 3 columns against the public data. Compare the three measurable rows against the production deployment. Either the numbers converge on the framework's predictions or the gap exposes which lever the framework misses at internet scale.

The synthetic dataset shipped with this paper made the math testable. The OSS validation extends the test to real workloads at scales the production deployment never reached. Anyone running the analysis is invited to publish the resulting numbers. The dataset format in `propagation_events.csv` is the canonical interchange schema for reporting back.

---

10. Conclusion

The sweet spot for managing multiple release branches sits at the intersection of three operational disciplines. Automate the synchronous propagation so latency drops from days to minutes. Order the synchronous propagation so disorder stays low. Observe the synchronous propagation so the human loop is closed without flooding developers with notifications.

A team operating in the sweet spot sees workload stay linear in `(B − 1) ∗ C` even as the branch matrix grows. A team outside the sweet spot sees workload compound through `M`, `PL`, `DDF`, and `DOC`. The difference between the two is the operational debt turning a 2019 missed delivery into a 2023 customer escape.

The accompanying synthetic dataset and the deterministic generator make every number in this paper reproducible. The supplementary document carries the full implementation detail for readers building a similar system. The companion repository at https://github.com/brettleehari/SyncForwardProp and the visual page at https://brettleehari.github.io/SyncForwardProp/ host the dataset, the code, and the rendered dashboards. There is room for improvement in every part of this framework. Use the dataset, run your own experiments, and publish your improvements.

References

[1] Emad Shihab, Christian Bird, Thomas Zimmermann, "The Effect of Branching Strategies on Software Quality," Empirical Software Engineering Conference, Microsoft Research, 2012.

[2] Jez Humble, David Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation," Addison-Wesley, 2010.

[3] Nicole Forsgren, Jez Humble, Gene Kim, "Accelerate: The Science of Lean Software and DevOps," IT Revolution Press, 2018.

[4] Paul Hammant, “Trunk-Based Development,” trunkbaseddevelopment.com, 2021.

[5] Martin Fowler, “Patterns for Managing Source Code Branches,” martinfowler.com, 2020.

[6] Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy, “Site Reliability Engineering: How Google Runs Production Systems,” O’Reilly, 2016.

[7] Martin Kleppmann, “Designing Data-Intensive Applications,” O’Reilly, 2017.

[8] Hariprasad Sudharshan, “The Degree of Disorder Factor: Hidden Cost of Parallel Branching,” 2024.

Companion artifacts

- `propagation_events.csv`, 500-event synthetic reference dataset
- `generate_dataset.py`, deterministic generator (seed 42)
- `validate_dataset.py`, distribution validator
- `build_dashboards.py`, dashboard image generator
- `Synchronous_Forward_Propagation_Supplementary.pdf`, implementation detail, message templates, complete event schema, sample queries, gap-analysis procedure, implementation roadmap

Repository

- GitHub: https://github.com/brettleehari/SyncForwardProp
- Visual companion: https://brettleehari.github.io/SyncForwardProp/