I need a summary PDF about this project in current state, can you send it?

A direct PDF cannot be sent, but the content below can be copied into a document editor (e.g., Word, Google Docs) and exported as PDF.

Helm YAML Sync Tool – Design Summary

This document summarizes the current design of a TypeScript-based tool for comparing and synchronizing Kubernetes Helm-related YAML files (values.yaml, chart.yaml, and other *.yaml files) across environments (e.g., UAT → PROD) in a GitOps workflow with Argo CD.[1]

1. Goal and Use Cases

The tool helps maintain multiple environment-specific Helm configurations while keeping them consistent and reviewable. Main use cases:

- Compare UAT vs. PROD values.yaml and selectively sync changes.
- Apply regex-based value transformations (e.g., DB URLs, hostnames) per environment.
- Enforce YAML formatting, ordering, and quoting rules for clean Git diffs.
- Generate large multi-file diff reports in HTML for browser-based review, then apply changes and commit manually.[2]

The project is valid, practical, and complements existing Helm/GitOps tooling by focusing specifically on safe, config-driven value synchronization between folders.[3]

2. High-Level Behavior

The tool operates on two folders per run:

- **source**: newer or reference configuration (e.g., UAT repo subtree).
- **dest**: environment to update (e.g., PROD repo subtree).

Main steps for a run:

1. Load a YAML config file describing source/dest paths, glob patterns, transforms, exclusions, ordering rules, and stop rules.
2. Recursively discover matching YAML files in both folders using configured include masks.[4]
3. For each file path:

    - If present in both: compute deep diff (source vs dest), apply exclusions, apply regex-based transformations, and build a preview of the new dest content.[5]
    - If present only in source: treat as **new file**, copy and format into dest according to ordering and quoting rules.
4. In **dry-run** mode: do not write any files; instead produce a summary and, optionally, an HTML report opened automatically in a browser.[2] [6]
5. In **apply** mode (no `--dry-run`): write all applicable changes and new files in dest, ready for manual `git diff`, commit, and Argo CD sync.[1] [3]

No git operations are performed by the tool itself; commits are intentionally left to the user after reviewing changes.

3. Configuration File

The main configuration file (YAML) defines behavior and rules.

3.1 Core Paths and Globs

```
source: "./uat"               # Source folder
dest: "./prod"                # Destination folder

include:                      # Files to process
  - "**/values.yaml"
  - "**/chart.yaml"
  - "charts/**/xxxx.yaml"
```

- Uses glob patterns to select files to compare and sync (values.yaml, chart.yaml, and custom paths).[4]
- Allows adding more patterns to cover additional YAML types as needed.

3.2 Exclusions and Transforms

```
excludes:
  - "replicaCount"
  - "image.tag"
  - "$.secrets[*].password"    # JSONPath-style for nested values

transforms:
  "database.url":
    - find: "uat-db\\.(.*)"
      replace: "prod-db.$1"
  "redis.host":
    - find: "uat-redis"
      replace: "prod-redis"
```

- **excludes**: paths that must never be synchronized (e.g., replica counts, certain image tags, secrets).[7]

- **transforms**: path-specific regex rewriting rules to adapt values between environments (URLs, hostnames, suffixes, etc.).

3.3 Output Formatting and Quoting

```
output:
  indent: 2          # YAML indent size
  quoteValues: true  # Only values (right side of `:`) are double-quoted
```

Behavior:

- All keys remain unquoted.
- All scalar string values are emitted as `"value"` with double quotes, ensuring consistent YAML for Git diffs and Helm templating.[5]
- Indentation is fixed (e.g., 2 spaces) for uniform formatting.[8]

3.4 Ordering Rules (Per Pattern)

Ordering rules define how keys should be arranged at top level and nested levels, per glob pattern.

Global fallback:

```
order:
  topLevel:
    - "image"
    - "service"
    - "*"
```

Per-pattern overrides:

```
rules:
  "**/values.yaml":
    topLevel:
      - "image"
      - "resources"
      - "service"
      - "ingress"
      - "*"
    nested:
      "resources":
        - "requests"
        - "limits"

  "**/chart.yaml":
    topLevel:
      - "name"
      - "version"
      - "description"
      - "maintainers"
```

```
      - "*"

  "charts/**/xxxx.yaml":
    topLevel:
      - "enabled"
      - "replicas"
      - "config"
      - "*"
    nested:
      "config":
        - "db"
        - "cache"
        - "logging"
```

Each file is matched against these globs to select the appropriate ordering; if none match, global `order` is used.[4]

4. Stop Rules and Safety

Stop rules are safety guards that cause the process to fail fast if dangerous changes are detected, especially for production.[3]

Stop rules are associated with file patterns and specific paths inside the YAML:

```
stopRules:
  "**/chart.yaml":
    - type: "semverMajor"
      path: "version"    # Block major SemVer bumps in chart.yaml

  "**/values.yaml":
    - type: "semverMajor"
      path: "image.tag"  # Block major SemVer bump of image tag
    - type: "numeric"
      path: "replicaCount"
      min: 2             # Reject scaling to <2 replicas
      max: 10            # Reject scaling above 10

  "charts/**/xxxx.yaml":
    - type: "regex"
      path: "database.url"
      regex: "prod-db-backup"
```

Implementation details:

- SemVer checks use a library like `semver` to compare `old` vs `new` and detect true major version changes (e.g., 1.9.7 → 2.0.0).[9] [10]
- Numeric rules validate thresholds (replica counts, resource limits).
- Regex rules block forbidden patterns (e.g., backup DB endpoints or test hosts in PROD).

If any stop rule is violated:

- In normal mode, the process exits with an error and lists all violations; no files are written.

- In dry-run mode, violations are shown in console and HTML report.

4.1 Force Override

Safety can be bypassed explicitly using `--force`, similar to force options in other GitOps tools.[11]

- `--force` disables all stop rules during that run.
- Violations are still reported as warnings, but do not block writes.
- Typical workflow:
  1. Run dry-run (no force) to see violations.
  2. Decide if changes are acceptable.
  3. Re-run with `--force` (no `--dry-run`) to apply.

5. Dry-Run, Apply, and Reporting

5.1 CLI Interface

Example commands:

```
# Preview only (no changes)
npx tsx src/sync.ts --config config.yaml --dry-run --html-report ./report.html

# Apply changes (no dry-run, no stop rule override)
npx tsx src/sync.ts --config config.yaml

# Apply changes with safety override (force)
npx tsx src/sync.ts --config config.yaml --force
```

- **dry-run**: required to see all changes safely before applying. No confirmation prompts; the user deliberately reruns without `--dry-run` to apply.
- **force**: explicit override for stop rules.[11]

5.2 Console Summary

Dry-run output:

- Number of scanned files.
- Count of changes per file, including new files.
- Count of excluded changes.
- Any stop rule violations.

Example:

- `15 files scanned, 47 changes, 8 excluded`
- New files listed explicitly (copied if not dry-run, or previewed if dry-run).

5.3 HTML Diff Report

For large changes, dry-run can generate a tabbed HTML report and open it in a browser automatically.[2]

Features:

- Per-file tabs (values.yaml, chart.yaml, xxxx.yaml).
- Side-by-side diff visualization using a library like `diff2html`.[12] [2]
- Optional grouping for:
  - New files
  - Updated files
  - Unchanged files
  - Stop rule violations / forced changes

HTML is written to a configurable path and opened automatically in the system browser for convenient review.[6]

6. New File Handling

If a YAML file exists in source but not in dest:

- It is treated as **new**.
- Its contents are loaded from source.
- Ordering rules and value quoting are applied according to the same glob-based logic used for existing files.
- In dry-run, the file appears as "NEW FILE" with the formatted output previewed.
- In apply mode, the file is created in dest with the formatted content.[7]

7. Implementation Stack (Current Plan)

- **Language**: TypeScript, executed directly (e.g., with `tsx`), fitting your existing TypeScript tooling preferences.[4]
- **Core libraries**:
  - YAML parsing/serialization: `js-yaml` or similar.[5]
  - Deep comparison: `deep-diff` or custom recursive diff.[5]
  - SemVer: `semver` for major change detection.[9]
  - Glob matching: `fast-glob` / `minimatch`.[4]
  - HTML diff: `diff2html` for visual diffs.[2]
  - Browser open: `open` package to launch the HTML report.[6]

The tool is designed to slot into an existing GitOps workflow: use dry-run and HTML reports to review multi-environment updates, then apply, review `git diff`, commit, and let Argo CD handle cluster synchronization.[1] [3]

If you want, the next step can be a concrete `config.yaml` example for your real UAT/PROD repos or a minimal TypeScript skeleton you can paste into your project.

⁂

1. projects.tooling_with_gitops
2. https://argo-cd.readthedocs.io/en/latest/user-guide/helm/
3. interests.devops_kubernetes
4. preferences.yaml_formatting
5. projects.kubernetes_values_yaml_diff_tool
6. https://www.npmjs.com/package/diff2html
7. https://cloud.google.com/kubernetes-engine/enterprise/config-sync/docs/concepts/gitops-best-practices
8. preferences.config_files
9. https://blog.logrocket.com/running-typescript-node-js-tsx-vs-ts-node-vs-native/
10. https://www.npmjs.com/package/@types/deep-diff
11. projects.file_processing_tool_requirements
12. https://stackoverflow.com/questions/62438559/open-html-report-in-a-browser
13. https://stackoverflow.com/questions/27820289/exclude-fields-from-json-using-jsonpath
14. https://spacelift.io/blog/yaml
15. https://www.npmjs.com/package/semver
16. https://semver.org
17. https://github.com/argoproj/argo-cd/discussions/5172
18. https://github.com/rtfpessoa/diff2html/blob/master/README.md