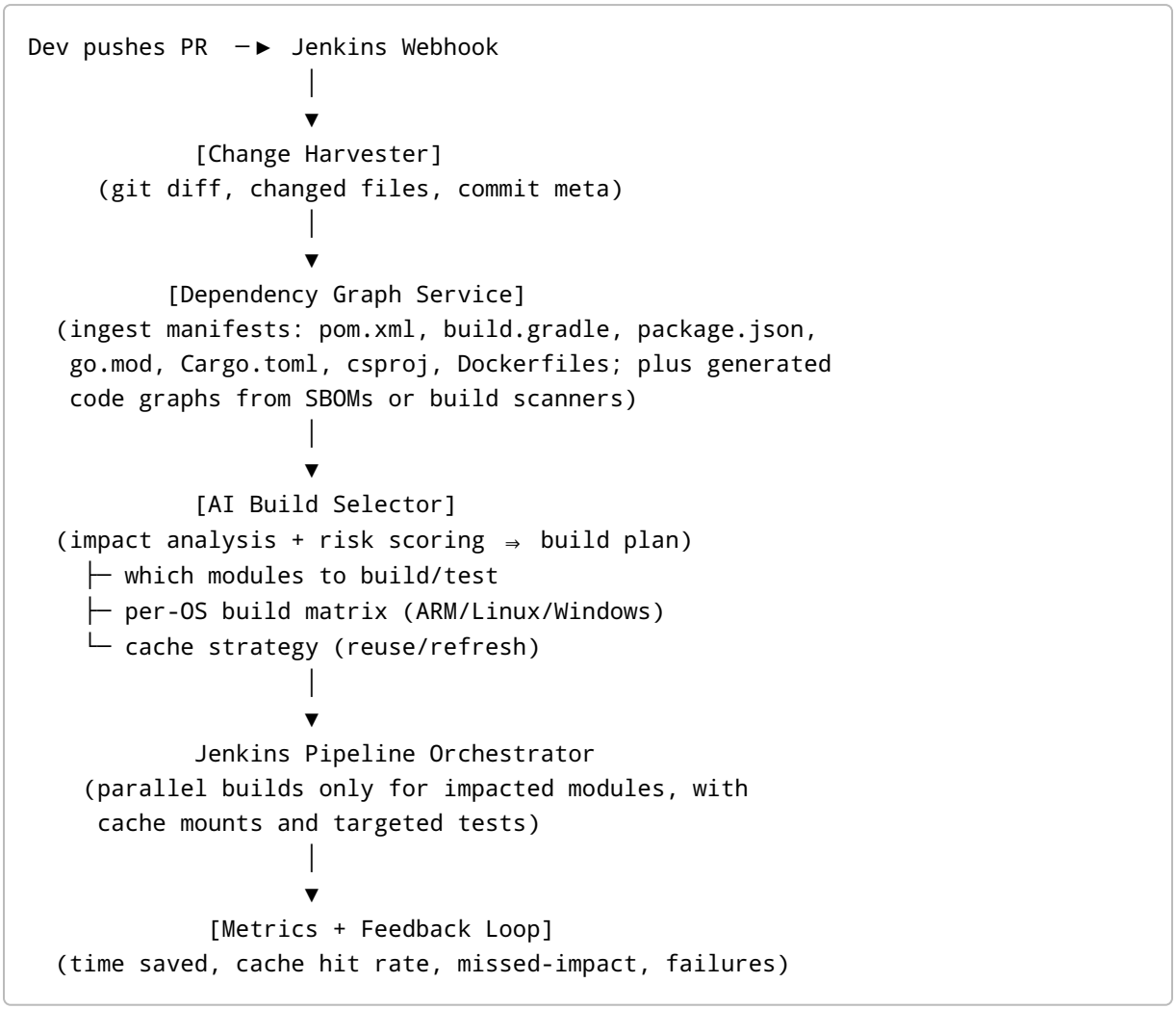
Step 1 – AI Build Optimization via Dependency Analysis

Goals

- **Skip unnecessary builds** and **shorten pipeline time** by only rebuilding modules/packages impacted by a change.
- **Prioritize** high-risk or high-fanout modules; **defer** or **parallelize** low-impact work.
- **Reuse caches** (Docker layers, Gradle/Maven/NPM caches) safely with change-aware invalidation.

High-Level Architecture



Key data sources: git diff, dependency manifests, SBOMs (Syft/Grype), historical build logs, test coverage map.

Components & Tools

- **Dependency extraction**
- JVM: `mvn dependency:tree -DoutputType=dot`, `gradle dependencies --scan`
- Node: parse `package-lock.json` / `pnpm-lock.yaml`, `npm ls --json`
- Python: `pipdeptree --json-tree`
- Go: `go mod graph`
- .NET: `dotnet list package --include-transitive`
- Containers: parse `Dockerfile` (base image, COPY paths), build SBOM via **Syft**
- **Graph store**: serialized as JSON, or Neo4j for large repos
- **AI/Heuristics**: start with rules; extend to XGBoost later
- **Jenkins**: Multibranch + Shared Library for `decideBuildPlan()`
- **Caching**: Docker BuildKit, remote Gradle cache, Maven local repo, npm cache, ccache for C/C++

Data Model

```
{
  "modules": [
    { "id": "svc-payment", "lang": "java", "os": ["linux", "arm"], "path":
"services/payment" },
    { "id": "web-portal", "lang": "node", "os": ["linux", "windows"], "path":
"apps/web" }
  ],
  "edges": [ { "from": "lib-core", "to": "svc-payment" }, { "from": "lib-ui",
"to": "web-portal" } ],
  "filesToModule": { "services/payment/src/...": "svc-payment", "apps/web/
src/...": "web-portal" },
  "testsMap": { "svc-payment": ["tests/payment/**"], "web-portal": ["apps/web/
tests/**"] }
}
```

Algorithm (Heuristic v1)

1. **Map changed files → owning modules** using `filesToModule` globs.
2. **Compute impact set**: all changed modules + downstream dependents via transitive closure on `edges`.
3. **Risk score** per impacted module:

4. `risk = w1*LOC + w2*fanout + w3*criticality + w4*dep_bump + w5*test_gap - w6*recent_success`
5. **Build plan**:
6. Always include highest risk modules.
7. For low-risk leaf modules with no runtime changes, **skip build** and reuse last artifact if inputs unchanged (content hash).
8. Derive **per-OS** matrix from module's supported OS set.
9. **Test selection**: run only mapped tests + smoke suite; full regression only if risk above threshold.

Jenkins Integration (Skeleton)

```
stage('Decide Build Plan') {
  steps {
    sh '''
      python3 tools/depgraph/build_graph.py --out depgraph.json
      python3 tools/build_selector.py
        --depgraph depgraph.json
        --git-range origin/main...HEAD
        --history .ci/history.json
        --out buildplan.json
    '''
    script {
      def plan = readJSON file: 'buildplan.json'
      env.BUILD_PLAN = writeJSON(returnText: true, json: plan)
    }
  }
}

stage('Parallel Impacted Builds') {
  steps {
    script {
      def plan = readJSON text: env.BUILD_PLAN
      def branches = [:]
      plan.modules.each { m ->
        branches["${m.id}-${m.os}"] = {
          node(m.os == 'windows' ? 'win' : 'linux') {
            checkout scm
            withEnv(["CACHE_KEY=${m.cacheKey}"]) {
              if (m.lang == 'java') {
                sh 'gradle build -x test --build-cache'
                sh 'gradle test --tests ' + m.tests.join(' ')
              } else if (m.lang == 'node') {
                sh 'npm ci && npm run build && npm test -- ' + m.tests.join(' ')
              }
              // archiveArtifacts, push to Artifactory per module
```

```
            }
          }
        }
      }
      parallel branches
    }
  }
}
```

Example: `tools/build_selector.py` (pseudo)

```
# inputs: depgraph.json, git diff, historical stats
# output: buildplan.json { modules: [ {id, os, lang, tests, cacheKey} ] }

import json, subprocess, hashlib

def changed_files():
    out = subprocess.check_output(['git', 'diff', '--name-only', 'origin/
main...HEAD']).decode().splitlines()
    return out

# compute impacted modules via reverse dependency traversal

# compute cacheKey: hash of module sources + lockfiles + Dockerfile chunks

def cache_key(paths):
    h = hashlib.sha256()
    for p in paths: h.update(open(p,'rb').read())
    return h.hexdigest()[:12]

# assign risk, choose OS targets, tests

# emit buildplan.json
```

Cache Strategy

- **Content-addressed keys**: combine module source hash + lockfile hash + toolchain version.
- **Gradle remote cache**: configure `org.gradle.caching=true` and shared cache bucket.
- **Docker**: BuildKit + `--cache-from` using last successful image for that module+OS.
- **Node**: `npm ci` with cached `~/.npm`; restore/save using Jenkins `stash/unstash` or cache plugin.

Validation & Guardrails

- Periodically force a **full build** (e.g., nightly) and compare outputs.
- **Missed-impact detector**: if a skipped module fails in downstream integration, record a miss → increase its risk weight.
- Track KPIs: pipeline duration, executor hours, cache hit %, false skips.

Rollout Plan (for Step 1)

1. Implement dependency extraction and graph JSON.
2. Ship `build_selector.py` with heuristics; dry-run mode that only **logs** skipped modules for 1–2 weeks.
3. Turn on selective builds for low-risk modules.
4. Add cache keys and remote caches.
5. Start collecting history to later upgrade to ML model.

Deliverables

- `tools/depgraph/build_graph.py` – parse manifests → `depgraph.json`
- `tools/build_selector.py` – select impacted modules → `buildplan.json`
- Jenkins shared lib function `decideBuildPlan()`
- Cache config for Gradle/Maven/Node/Docker
- Dashboards for **time saved**, **cache hit rate**, **false-skip rate**