GitVersion Shell Script

A lightweight shell implementation of GitVersion that automatically generates semantic version numbers from your Git repository history.

Features

- **Automatic Semantic Versioning**: Calculates version numbers based on Git history and branch structure
- **Multiple Workflow Support**: GitFlow, GitHubFlow, and trunk-based development workflows
- **Commit Message Parsing**: Detects version increments from conventional commit messages
- **Branch-Aware Versioning**: Different versioning strategies for main, develop, feature, release, and hotfix branches
- **Flexible Output**: Support for both human-readable text and structured JSON output
- **Pre-release Versions**: Automatic generation of alpha, beta, and feature-specific pre-release versions
- **Build Metadata**: Includes commit count and SHA information

Installation

```
# Download and make executable
curl -o gitversion.sh
https://raw.githubusercontent.com/VirtuallyScott/battle-tested-
devops/main/gitversion-sh/gitversion.sh
chmod +x gitversion.sh

# Optionally, move to PATH
sudo mv gitversion.sh /usr/local/bin/gitversion
```

Usage

Basic Usage

```
# Calculate version for current branch
./gitversion.sh

# Output: 1.2.3+5+abc1234
```

Command Line Options

```
gitversion [OPTIONS]

OPTIONS:
```

```
    -h, --help              Show help message
    -v, --version           Show version information
    -o, --output FORMAT     Output format
(json|text|AssemblySemVer|AssemblySemFileVer) [default: text]
    -c, --config FILE       Path to configuration file
    -b, --branch BRANCH     Target branch [default: current branch]
    -w, --workflow TYPE     Workflow type (gitflow|githubflow|trunk)
[default: gitflow]
    --major                 Force major version increment
    --minor                 Force minor version increment
    --patch                 Force patch version increment
    --next-version VERSION  Override next version
```

Examples

```
# Basic version calculation
gitversion

# JSON output for CI/CD integration
gitversion -o json

# Output AssemblySemVer format (1.2.3.0)
gitversion -o AssemblySemVer

# Output AssemblySemFileVer format (1.2.3.0)
gitversion -o AssemblySemFileVer

# Calculate version for specific branch
gitversion -b main

# Force major version increment
gitversion --major

# Use GitHub Flow workflow
gitversion -w githubflow

# Override next version
gitversion --next-version 2.0.0
```

Workflows

GitFlow (Default)

Perfect for projects using the GitFlow branching model:

- **main/master**: Stable releases (1.0.0)
- **develop**: Development versions (1.1.0-alpha.5)
- **feature/***: Feature branches (1.1.0-feature-name.3)
- **release/***: Release candidates (1.1.0-beta.2)

- **hotfix/***: Hotfix versions (1.0.1-hotfix.1)

GitHubFlow

Simplified workflow for GitHub-style development:

- **main/master**: Stable releases
- **feature branches**: Pre-release versions with branch name

Trunk-based

All branches treated as main branch versions.

GitFlow Workflow Best Practices

Understanding GitFlow Branches

GitFlow is a branching model that defines specific branch types for different stages of development:

```
main/master    ←── Production-ready releases (1.0.0, 1.1.0, 2.0.0)
     ↑
   hotfix/*     ←── Critical production fixes (1.0.1-hotfix.1)
     ↑
   release/*    ←── Release preparation (1.1.0-beta.1, 1.1.0-beta.2)
     ↑
   develop      ←── Integration branch (1.1.0-alpha.15)
     ↑
   feature/*    ←── New features (1.1.0-user-auth.5)
```

Branch Versioning Strategy

Branch Type	Version Pattern	Example	Purpose
main	`X.Y.Z`	`1.0.0`	Production releases
develop	`X.Y.Z-alpha.N`	`1.1.0-alpha.15`	Development integration
feature/*	`X.Y.Z-{name}.N`	`1.1.0-user-auth.5`	Feature development
release/*	`X.Y.Z-beta.N`	`1.1.0-beta.2`	Release candidates
hotfix/*	`X.Y.Z-hotfix.N`	`1.0.1-hotfix.1`	Production fixes

Complete Release Management Workflow

1. Starting a New Release

```
# From develop branch - start release preparation
git checkout develop
git pull origin develop
```

```
# Create release branch
git checkout -b release/v1.2.0

# Check the version - should show beta pre-release
./gitversion.sh
# Output: 1.2.0-beta.1+0+abc1234

# Push release branch
git push -u origin release/v1.2.0
```

2. Release Branch Development

```
# On release/v1.2.0 branch
git checkout release/v1.2.0

# Make release-specific changes (bug fixes, documentation)
echo "Release notes for v1.2.0" > RELEASE_NOTES.md
git add RELEASE_NOTES.md
git commit -m "docs: add release notes for v1.2.0"

# Check updated version
./gitversion.sh
# Output: 1.2.0-beta.2+1+def5678

# Continue with more release preparation
git commit -m "fix: resolve minor UI issue +semver: patch"
./gitversion.sh
# Output: 1.2.0-beta.3+2+ghi9012
```

3. Finalizing the Release

```
# When ready to release, merge to main
git checkout main
git pull origin main
git merge --no-ff release/v1.2.0 -m "Release v1.2.0"

# Check final version on main
./gitversion.sh
# Output: 1.2.0+0+jkl3456

# Tag the release
VERSION=$(./gitversion.sh)
git tag -a "v$VERSION" -m "Release v$VERSION"
git push origin main --tags
```

4. Back-merge to Develop

```
# Merge main back to develop to sync changes
git checkout develop
git pull origin develop
git merge --no-ff main -m "Merge main after v1.2.0 release"
git push origin develop

# Clean up release branch (optional)
git branch -d release/v1.2.0
git push origin --delete release/v1.2.0
```

Hotfix Workflow

1. Creating a Hotfix

```
# Critical bug found in production (v1.2.0)
git checkout main
git pull origin main
git checkout -b hotfix/critical-security-fix

# Check hotfix version
./gitversion.sh
# Output: 1.2.1-hotfix.1+0+mno7890

# Make the fix
git commit -m "fix: resolve security vulnerability +semver: patch"
./gitversion.sh
# Output: 1.2.1-hotfix.2+1+pqr2345
```

2. Deploying the Hotfix

```
# Merge to main
git checkout main
git merge --no-ff hotfix/critical-security-fix -m "Hotfix v1.2.1"

# Check final version
./gitversion.sh
# Output: 1.2.1+0+stu6789

# Tag and push
VERSION=$(./gitversion.sh)
git tag -a "v$VERSION" -m "Hotfix v$VERSION"
git push origin main --tags

# Back-merge to develop
git checkout develop
git merge --no-ff main -m "Merge hotfix v1.2.1 to develop"
git push origin develop
```

```
# Clean up
git branch -d hotfix/critical-security-fix
git push origin --delete hotfix/critical-security-fix
```

Feature Development Workflow

1. Feature Branch Creation

```
# Start new feature from develop
git checkout develop
git pull origin develop
git checkout -b feature/user-authentication

# Check initial version
./gitversion.sh
# Output: 1.3.0-user-authentication.1+0+vwx1234

# Develop the feature
git commit -m "feat: add user login endpoint"
git commit -m "feat: add password validation"
git commit -m "test: add authentication tests"

# Check final feature version
./gitversion.sh
# Output: 1.3.0-user-authentication.4+3+yza5678
```

2. Feature Integration

```
# When feature is complete, merge to develop
git checkout develop
git pull origin develop
git merge --no-ff feature/user-authentication -m "feat: integrate user
authentication"

# Check develop version after merge
./gitversion.sh
# Output: 1.3.0-alpha.25+24+bcd9012

# Clean up feature branch
git branch -d feature/user-authentication
git push origin --delete feature/user-authentication
```

Long-Running Release Branches

Release branches can remain available indefinitely for maintenance and patch releases:

Maintaining Multiple Release Versions

```
# Main development continues on develop
git checkout develop
./gitversion.sh
# Output: 2.0.0-alpha.45+44+efg3456

# Meanwhile, maintain v1.2.x series
git checkout release/v1.2.0
git checkout -b release/v1.2.x  # Long-term maintenance branch

# Apply backported fixes
git cherry-pick abc1234  # Cherry-pick from main or develop
git commit -m "fix: backport security patch +semver: patch"

# Check maintenance version
./gitversion.sh
# Output: 1.2.1-beta.1+1+hij7890

# Release maintenance version
git checkout main-v1.2.x  # Separate main branch for v1.2.x line
git merge --no-ff release/v1.2.x -m "Release v1.2.1"

VERSION=$(./gitversion.sh)
git tag -a "v$VERSION" -m "Maintenance release v$VERSION"
```

Configuration for GitFlow

Create a comprehensive `GitVersion.yml` for GitFlow:

```
next-version: '1.0.0'
branches:
  main:
    increment: Patch
    prevent-increment-of-merged-branch-version: true
    track-merge-target: false
    regex: '^master$|^main$'
    source-branches: ['develop', 'release', 'hotfix']

  develop:
    increment: Minor
    prevent-increment-of-merged-branch-version: false
    track-merge-target: true
    regex: '^develop$'
    source-branches: []
    pre-release-tag: 'alpha'

  feature:
    increment: Minor
    prevent-increment-of-merged-branch-version: false
```

```
    track-merge-target: false
    regex: '^features?[/-]'
    source-branches: ['develop']
    pre-release-tag: '{BranchName}'

  release:
    increment: None
    prevent-increment-of-merged-branch-version: true
    track-merge-target: false
    regex: '^releases?[/-]'
    source-branches: ['develop']
    pre-release-tag: 'beta'

  hotfix:
    increment: Patch
    prevent-increment-of-merged-branch-version: false
    track-merge-target: false
    regex: '^hotfix(es)?[/-]'
    source-branches: ['main']
    pre-release-tag: 'hotfix'

commit-message-incrementing:
  enabled: true
  increment-mode: Enabled

ignore:
  sha: []

merge-message-formats: {}
```

Best Practices Summary

1. **Never commit directly to main** - Always use pull requests from release/hotfix branches
2. **Keep release branches** - Don't delete them; they serve as historical markers
3. **Use semantic commit messages** - Enable automatic version detection
4. **Tag all releases** - Use the version from gitversion.sh for consistent tagging
5. **Back-merge to develop** - Ensure develop stays in sync with production fixes
6. **Version early and often** - Check versions before major operations
7. **Document breaking changes** - Use conventional commits and BREAKING CHANGE footers

Version Increment Detection

The script automatically detects version increments from commit messages:

Semantic Version Tags

Add these tags to commit messages to control version increments:

```
git commit -m "fix: resolve login issue +semver: patch"
git commit -m "feat: add user profiles +semver: minor"
```

```
git commit -m "feat!: redesign API +semver: major"
```

Conventional Commits

The script also recognizes conventional commit patterns:

- `feat:` → Minor increment
- `feat!:` → Major increment (breaking change)
- `fix:` → Patch increment
- `BREAKING CHANGE:` → Major increment

Configuration

Configuration Files

GitVersion.sh supports both JSON and YAML configuration files for advanced customization.

JSON Configuration (GitVersion.json)

```
{
  "next-version": "1.0.0",
  "branches": {
    "main": {
      "increment": "Patch",
      "tag": "stable"
    },
    "develop": {
      "increment": "Minor",
      "tag": "alpha"
    },
    "feature": {
      "increment": "Minor",
      "tag": "feat"
    },
    "release": {
      "increment": "None",
      "tag": "beta"
    },
    "hotfix": {
      "increment": "Patch",
      "tag": "hotfix"
    }
  },
  "ignore": {
    "sha": []
  },
  "merge-message-formats": {},
  "commit-message-incrementing": {
    "enabled": true,
    "increment-mode": "Enabled"
```

```
    }
}
```

YAML Configuration (GitVersion.yml)

```yaml
next-version: '1.0.0'

branches:
  main:
    increment: Patch
    tag: stable
    regex: '^master$|^main$'
    source-branches: ['develop', 'release', 'hotfix']

  develop:
    increment: Minor
    tag: alpha
    regex: '^develop$'
    source-branches: []

  feature:
    increment: Minor
    tag: feat
    regex: '^features?[/-]'
    source-branches: ['develop']

  release:
    increment: None
    tag: beta
    regex: '^releases?[/-]'
    source-branches: ['develop']

  hotfix:
    increment: Patch
    tag: hotfix
    regex: '^hotfix(es)?[/-]'
    source-branches: ['main']

ignore:
  sha: []

merge-message-formats: {}

commit-message-incrementing:
  enabled: true
  increment-mode: Enabled
```

Configuration Usage

```
# Use JSON configuration
./gitversion.sh --config GitVersion.json

# Use YAML configuration
./gitversion.sh --config GitVersion.yml

# Configuration with specific branch
./gitversion.sh --config GitVersion.yml --branch develop

# Override configuration with CLI args
./gitversion.sh --config GitVersion.yml --major --output json
```

Advanced Configuration Examples

Multi-Environment Configuration

```
# GitVersion.yml for multiple environments
next-version: '1.0.0'

branches:
  main:
    increment: Patch
    tag: ''
    regex: '^master$|^main$'

  develop:
    increment: Minor
    tag: 'dev'
    regex: '^develop$'

  'release/staging':
    increment: None
    tag: 'staging'
    regex: '^release/staging$'

  'release/production':
    increment: None
    tag: 'rc'
    regex: '^release/production$'

  feature:
    increment: Minor
    tag: '{BranchName}'
    regex: '^feature[/-]'

  hotfix:
    increment: Patch
    tag: 'hotfix'
    regex: '^hotfix[/-]'
```

```
commit-message-incrementing:
  enabled: true
  increment-mode: Enabled
```

Monorepo Configuration

```
{
  "next-version": "1.0.0",
  "branches": {
    "main": {
      "increment": "Minor",
      "tag": ""
    },
    "develop": {
      "increment": "Minor",
      "tag": "alpha"
    },
    "feature": {
      "increment": "Minor",
      "tag": "{BranchName}"
    }
  },
  "ignore": {
    "sha": [
      "a4a7dce85cf63874e984719b4b25a0cc2a6b46dd",
      "b769ef8b7a2c4a5d0f36b28c4b6a8e3f5c8d9f2a"
    ]
  },
  "commit-message-incrementing": {
    "enabled": true,
    "increment-mode": "Enabled"
  }
}
```

Configuration Validation

Test your configuration before applying:

```
# Validate configuration syntax
./gitversion.sh --config GitVersion.yml --output json | jq .

# Test different branches with configuration
./gitversion.sh --config GitVersion.yml --branch main
./gitversion.sh --config GitVersion.yml --branch develop
./gitversion.sh --config GitVersion.yml --branch feature/test

# Debug configuration loading
DEBUG=true ./gitversion.sh --config GitVersion.yml
```

Environment Variables

```
# Set default workflow
export GITVERSION_WORKFLOW=githubflow

# Enable debug logging
export DEBUG=true

# Set default output format
export GITVERSION_OUTPUT=json

# Set default config file
export GITVERSION_CONFIG=GitVersion.yml
```

Output Formats

Text Output (Default)

```
1.2.3-alpha.5+10+abc1234
```

Assembly Version Outputs

AssemblySemVer

```
1.2.3.0
```

The `AssemblySemVer` format outputs a four-part version number suitable for .NET Assembly versioning, always ending with `.0` for the build number.

AssemblySemFileVer

```
1.2.3.0
```

The `AssemblySemFileVer` format outputs a four-part version number suitable for .NET Assembly file versioning, always ending with `.0` for the build number.

JSON Output

```
{
  "Major": 1,
  "Minor": 2,
```

```
  "Patch": 3,
  "PreReleaseTag": "alpha.5",
  "PreReleaseTagWithDash": "-alpha.5",
  "BuildMetaData": "10+abc1234",
  "BuildMetaDataPadded": "+10+abc1234",
  "FullBuildMetaData": "10+abc1234",
  "MajorMinorPatch": "1.2.3",
  "SemVer": "1.2.3-alpha.5+10+abc1234",
  "AssemblySemVer": "1.2.3.0",
  "AssemblySemFileVer": "1.2.3.0",
  "FullSemVer": "1.2.3-alpha.5+10+abc1234",
  "InformationalVersion": "1.2.3-alpha.5+10+abc1234",
  "BranchName": "develop",
  "EscapedBranchName": "develop",
  "Sha": "abc1234567890def",
  "ShortSha": "abc1234",
  "NuGetVersionV2": "1.2.3-alpha.5+10+abc1234",
  "NuGetVersion": "1.2.3-alpha.5+10+abc1234",
  "VersionSourceSha": "abc1234567890def",
  "CommitsSinceVersionSource": 10,
  "CommitDate": "2025-01-15 10:30:45 +0000"
}
```


CI/CD Integration

GitHub Actions

```yaml
- name: Calculate Version
  id: version
  run: |
    VERSION=$(./gitversion.sh)
    echo "version=$VERSION" >> $GITHUB_OUTPUT

- name: Build and Tag
  run: |
    docker build -t myapp:${{ steps.version.outputs.version }} .
```

GitLab CI

```yaml
version:
  script:
    - VERSION=$(./gitversion.sh)
    - echo "VERSION=$VERSION" >> build.env
  artifacts:
    reports:
      dotenv: build.env
```

Jenkins

```
pipeline {
    stages {
        stage('Version') {
            steps {
                script {
                    def version = sh(script: './gitversion.sh',
returnStdout: true).trim()
                    env.VERSION = version
                }
            }
        }
    }
}
```

Branch Strategies

Feature Branches

```
# On feature/user-auth branch
gitversion
# Output: 1.1.0-user-auth.3+15+def5678
```

Release Branches

```
# On release/1.2.0 branch
gitversion
# Output: 1.2.0-beta.2+8+ghi9012
```

Hotfix Branches

```
# On hotfix/critical-fix branch
gitversion
# Output: 1.1.1-hotfix.1+2+jkl3456
```

Troubleshooting

Debug Mode

Enable debug logging to see how versions are calculated:

```
DEBUG=true ./gitversion.sh
```

Common Issues

1. **No version tags found**: The script starts from 0.0.0 if no semantic version tags exist
2. **Invalid tag format**: Ensure tags follow semantic versioning (v1.2.3 or 1.2.3)
3. **Not a git repository**: Run the script from within a git repository
4. **Parsing errors**: Check that commit messages don't contain invalid characters

Validation

Test version calculation without making changes:

```
# Test different scenarios
gitversion -b main
gitversion -b develop
gitversion --major
gitversion --next-version 2.0.0
```

Testing and Validation

GitVersion.sh includes a comprehensive test suite to ensure compatibility with GitTools/GitVersion:

Running Tests

```
# Run all test suites
cd gitversion-sh/tests
./run_tests.sh

# Run specific test suites
./test_gitversion_compatibility.sh
./tmp/config_file_test/test_config_files.sh
./tmp/semver_rules_test/test_semver_rules.sh
```

Test Coverage

The test suite validates:

- **GitVersion Compatibility**: Ensures behavior matches GitTools/GitVersion
- **Configuration Parsing**: Tests JSON and YAML configuration files
- **Semantic Versioning Rules**: Tests conventional commit patterns and +semver tags
- **Branch Strategies**: Tests GitFlow, GitHub Flow workflows
- **Output Formats**: Validates JSON structure and Assembly version formats
- **Error Handling**: Tests invalid inputs and edge cases

Manual Testing Scenarios

```
# Test version calculation across workflows
./gitversion.sh --workflow gitflow --branch main
./gitversion.sh --workflow githubflow --branch feature/test
./gitversion.sh --workflow trunk --branch any-branch

# Test configuration precedence
./gitversion.sh --config GitVersion.yml --major --next-version 3.0.0

# Test commit message parsing
git commit -m "feat: new feature +semver: minor"
./gitversion.sh

git commit -m "fix: bug fix

BREAKING CHANGE: API changed"
./gitversion.sh
```

Integration Testing

Test with your CI/CD pipeline:

```
# GitHub Actions example
name: Version Validation
on: [push]
jobs:
  version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0  # Required for GitVersion

      - name: Test Version Calculation
        run: |
          chmod +x gitversion-sh/gitversion.sh
          VERSION=$(./gitversion-sh/gitversion.sh)
          echo "Calculated version: $VERSION"

          # Validate version format
          if [[ ! "$VERSION" =~ ^[0-9]+\.[0-9]+\.[0-9]+ ]]; then
            echo "Invalid version format: $VERSION"
            exit 1
          fi

          # Test JSON output
          JSON=$(./gitversion-sh/gitversion.sh --output json)
          echo "$JSON" | jq . > /dev/null

      - name: Test Configuration
        run: |
          # Test with configuration file
```

```
          cat > GitVersion.yml << EOF
          next-version: '1.0.0'
          branches:
            main:
              increment: Patch
          EOF

          ./gitversion-sh/gitversion.sh --config GitVersion.yml
```

Compatibility

- **Shell**: Bash 4.0+ (uses associative arrays)
- **Git**: 2.0+ (uses modern git commands)
- **OS**: Linux, macOS, Windows (with Git Bash/WSL)
- **Dependencies**:
  - `jq` (optional, for JSON configuration files)
  - `yq` (optional, for YAML configuration files)

Contributing

1. Fork the repository
2. Create a feature branch
3. Make your changes
4. Add tests for new functionality
5. Submit a pull request

License

This project is licensed under the MIT License - see the LICENSE file for details.

Acknowledgments

- Inspired by the original GitVersion project
- Follows Semantic Versioning specifications
- Compatible with Conventional Commits