DZone Refcardz

Brought to you by: CollabNet.

Git Patterns and Anti-Patterns

Scaling from Workgroup to Enterprise

By: Luca Milanesio

Git is the most popular DVCS (Distributed Version Control System) in terms of market adoption. Grasping Git adequately, however, can be difficult. This Refcard is designed to help you transition to Git with confidence and understanding, and is especially suited to enterprise-scale application development. This card assumes familiarity with version control systems (VCS) in general, and at least passing familiarity with Git. For Git basics, see Refcard #94: Getting Started with Git.)

GIT CHALLENGES

Most teams considering Git are already using some kind of VCS (e.g. Subversion, CVS or ClearCase). These teams face two key challenges.

Challenge #1: Migrating Concepts and Terms

DVCS makes you think differently. The concepts and terms proper to DVCS are deeply different from those proper to a traditional, centralized VCS. Translate these terms and concepts immediately.

Challenge #2: Migrating Team Development Practices

DVCS makes teams work together differently. As a rule, developers resist changes imposed by management – especially when the changes affect the whole development process as significantly as the switch to DVCS does. Furthermore, developers and managers may both feel that the 'deregulation' permitted by Git's local branches will (for the developers) produce lousy code or (for the managers) introduce security risks, compliance risks, and opacity. All of these concerns are legitimate, and all real and imagined problems need to be addressed.

GIT PATTERNS AND ANTI-PATTERNS: OVERVIEW

This Refcard takes a page from the object-oriented developer's book and presents sixteen reusable solutions to common Git adoption problems in the familiar form of Patterns and Anti-Patterns.

Pattern	Description
#1. Step-wise migration	Don't migrate everything at once .
#2. Git champions	Acclimate your teams to DVCS-thinking by clustering migrations around champions.
#3. CLI first	Keep Git's guts transparent and use tools only after everyone knows how the Git CLI works.
#4. Cheat sheets	Make sure cheat sheets are readily available, even after initial adoption.
#5. Blessed repository	For early adoption, and later for small and medium teams, a shared 'blessed' repository will ease developers into Git and simplify CI.
#6. Replication	For large enterprises, the single 'blessed' repo pattern makes less sense.
#7. Published branch strategy	Avoid anarchy, but enforce order via a well-understood strategy, not a restrictive VCS.
#8. Per-feature topic branches	Define one topic branch for each individual feature.
#9. Local-only rebasing	Limit rebasing to local repositories or individual branches only.
#10. Branch-level permissions	Define one set of permissions for experimental branches and another for stable release branches.
#11. Git protocol strategy	Enforce compatibility of Git protocols with your ICT security standards.
#12. Identity enforcement via user registry	Make sure everyone is fully accountable for all commits.

PATTERN #1: STEP-WISE MIGRATION

Pattern	Migrate using these small, safe steps: Define scope > Migrate branches > Migrate infrastructure > Set cutover date > Commit to Git
Anti-pattern	Migrate everything at once, defining a freeze period until the migration is completed

DON'T migrate your code in a single step. This can sound easy at first, but poorly planned initial migration can damage your repos and branches considerably.

Step 1: Define migration scope correctly

Take some time to define the scope of what is being migrated to Git. Rarely migrate "everything" – a large mass of stable code history, happily residing in your current VCS, will just weigh down what should be a lean tool. Basically:

DON'T migrate dead repositories and branches (unless you have independent reasons to decommission your legacy VCS, e.g. licensing restrictions)

DO wield a judicious change-depth knife, carefully defining a maximum depth of changes in your branches' history to migrate

Remember that the migration process itself has to physically create every intermediate file status and store all snapshot trees in the Git repository. This operation may take a considerable amount of time, and large chunks of change history may be unhelpful, given your current pipeline.

Step 2: Migrate branches

Here's some good news: Git provides out-of-the-box commands for migrating from Subversion (git svn) and CVS (git cvsimport). And now the bad news: these out-of-the-box commands can take a long time to run, so you may need to repeat the process if you don't want to block the productivity of your team.

DO use scripts to run the migration again and again. This will (a) help you predict execution times and (b) fine-tune the list of branches and repositories that you intend to migrate.

Step 3: Migrate build infrastructure

Continuous Integration (CI) and Continuous Delivery (CD) frameworks like Jenkins makes are not tied to any particular VCS. But adding a VCS to your release mix can tangle CI / CD processes significantly. For example, your release process may contain a lot of existing VCS hardcoded commands that need to be changed manually.

DO duplicate existing scripts and declare a script freeze on the original versions until the Git migration is completed.

Step 4: Define a cutover date

Rigorously avoid cutover creep on a project-by-project basis.

DON'T force the entire company into a single cutover date. It's much better to start with a few projects, get feedback on the transition while other projects are still getting ready to migrate, and improve the process for projects with later cutover dates.

DO set migration projects to read-only at cutover so that developers can then commit to Git only. Otherwise, you'll risk stranding a few straggler-projects in an old release paradigm.

Furthermore, the early-migrating teams will have developed expertise and confidence with Git when the later-migrating teams are just getting started – which will help the earlier team-members serve as Git champions (see Pattern #3).

Step 5: Commit to Git!

But just in case:

DO keep a backup of your existing VCS system and its build running until all projects are running on Git flawlessly. Migrating once is hard enough and more depressing when the direction is backward.

DEFINE your roll-back procedure – you may run into unpredictable problems, such as people misusing the tool or with your Git Server or Build stability, and you must not let this interfere with your productivity. Insofar as the first Git migration attempt damages productivity, the next migration attempt is unlikely to happen.

Hybrid VCS

Throughout the migration process you need to manage more than one system at a time (i.e. legacy Subversion and the new Git repositories). You may consider as well, though not suggested, to keep some of the projects in your legacy VCS, because of their complex project history or their rather excessive cost to migrate consolidated (or end-of-life) project maintenance to a different system.

DO use proper tools (i.e. CollabNet TeamForge or SubGit) to make sure that you are able to manage both tools during migration in a consistent way.

The Bottom Line: Avoid 'big-bang' migration

Doing everything at once will not make things simpler, even though you may think that people can use local code for a couple of days.

Productivity will almost certainly suffer if you migrate everything at once. If you migrate everything at once, you won't learn anything from early iterations. No teams will gain significantly more Git experience than others (except on local branches, which won't help much).

And worse: if you migrate everything at once and something doesn't work right, you'll never know what went wrong.

PATTERN #2: GIT CHAMPIONS

Pattern	Let Git Champions spread DVCS culture throughout your organization by starting with a small, experimental project, and eventually spreading to common code.
Anti-pattern	Try to bring everyone up to speed in one fell swoop.

Changing the culture of your developers is usually the hardest part of the migration process. Months after your migration cutover date, you might still hear people commenting that using the previous VCS was easier. Buy-in is always crucial, but especially for DVCS; and when buy-in is crucial, plain-vanilla tech training is not enough.

DO reward buy-in from motivated people who will experiment with the new tools before distribution to a wider audience.

DO choose the most dynamic and innovative members of your team to drive the change. Give them time to understand the new concepts in detail – let them earn the title of 'Git champions'.

DO advertise your Git champions throughout the company. Enterprise-wide backing will magnify your Git champions' effectiveness.

DO distribute your Git champions among different locations and teams – for DVCS, local support, and from developers on the same project/team, this is essential.

DON'T underestimate the complexity and danger of Git. It isn't reasonable to expect everyone to learn all 150+ commands from the documentation alone. For a tool this complex, the more local the support, the better. Also consider commercial support: http://www.collab.net/support/support-programs#git

The Bottom Line: Never let inexperienced developers run with Git by themselves

When you're coding for fun, learn to walk by falling and getting back up. When you're coding for enterprise-level profit, leaving Git in the hands of inexperienced developers is like abandoning a toddler with a dagger in its hands.

For a frightful appearance of this anti-pattern, consider this 2012 incident, wherein Eclipse experienced a disastrous branch loss due to a wrong Git command syntax: https://bugs.eclipse.org/bugs/show_bug.cgi?id=361707

There are management tools available that ease the migration to Git, and help enforce security and control. A good overview is at www.collab.net/git

PATTERN#3: CLI FIRST

Pattern	Use the command line first —no tools.
Anti-pattern	Use GUI and integrated tools first 'to save time' (and learn nothing about DVCS)

The path to learning how to use a DVCS effectively is long and difficult. Distributed systems are inherently complicated: you need to manage consistency between different states.

Drive your first DVCS as close to the metal as possible. Use the command line only, until you know second-nature how Git works.

Start with the simplest Git commands:

- **Git identity settings**
  (git config user.name, git config user.email)
- **Git local and remote operations**
  (git push, git fetch, git pull)

- **Searching through Git history**
  (git log, git diff, git bisect, git grep)
- **(Interactive) rebase**
  (git rebase, git rebase -i)

At this stage, tools may come in handy to increase productivity. Tools also may help to:

- **Display your Git branch network topology**
- **Display files' status (untracked, changed, stages, committed) as decoration on your file icons**
- **Perform two- or three-way file merges**

The Bottom Line: Learn to feel how DVCS works by learning to think exactly as Git thinks

You may not be surprised to learn that the most popular tool for Subversion is TortoiseSVN (http://tortoisesvn.net) – which makes version control pretty much drag-and-drop file copy. So there's a good chance that newly-minted Git users will expect to use similar tools for Git. But drag-and-drop file copy is nothing like using DVCS.

PATTERN#4: CHEAT SHEETS

Pattern	Use Git cheat sheets, including equivalence tables and pre-defined formulae.
Anti-pattern	Refer to the full git documentation, or search/browse through huge and confusing lists.

Developers with no knowledge of Git and the underlying concepts of DVCS will be heavily reliant on cheat sheets – and that's okay. So:

PUBLICIZE pre-defined "git formulas" for common tasks (e.g. http://refcardz.dzone.com/refcardz/getting-started-git, or http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf).

If you're migrating from Subversion, use the following equivalence table:

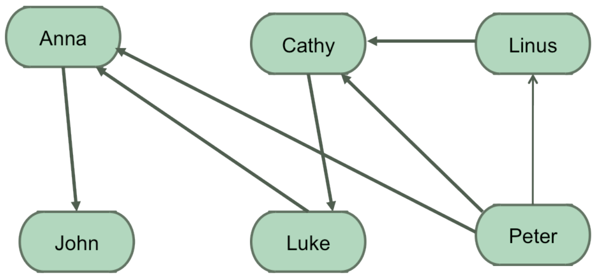
Subversion / Git equivalence table

Task	Subversion	Git
Repository creation	svnadmin create repo	git init
Checkout Server branch	svn co url/branch	git clone url branch
File management	svn add file svn rm file svn mv file	git add file git rm file git mv file
Commit to Server	svn commit	git commit -a git push
Update from Server	svn up	git stash git pull 'rebase git stash pop
Switch branches	svn switch url/branch	git checkout branch
Local files status	svn status	git status
Revert to latest version	svn revert path	git checkout path
Display file history	svn log path	git log path
Show annotated file	svn blame path	git blame path
Show file differences	svn diff 'rrev path	git diff rev path
Create branch	svn copy url1 branch	git branch initialrev
Create tag	svn copy url1 tagname	git tag tagname

GIT REPOSITORY PATTERNS: SMALL VS. MEDIUM VS. LARGE TEAMS

As developers grow more confident with Git, they will be eager to start taking the 'distributed' part seriously – i.e., the peer-to-peer and replication capabilities.

For small, local workgroups (up to 5-6 people), ungoverned peer-to-peer collaboration may work well. Questions like "who did the last pull?" can easily be answered, and resulting problems solved, during daily, face-to-face stand-ups. The dev-social graph is simple:



But as the dev-social graph gets more complex, peer-to-peer and replication capabilities can quickly get dangerous. **To keep Git safe for medium and larger teams, choose one of the following two patterns.**

PATTERN#5: BLESSED REPOSITORY (FOR MEDIUM-SIZED TEAMS)

Pattern	Use Git with a shared but 'blessed' repository.
Anti-pattern	Allow developers to publish their repository in a many-to-many pull exchange fashion.

When teams increase in size and are distributed among different locations, or when you need to integrate with an ecosystem of Continuous Integration (CI) / Continuous Delivery (CD) for builds, tests and deployments, it is key to define and publish one repository – the '**blessed repository**' – as a single point of reference for the entire team.

PATTERN#6: REPLICATION (FOR LARGE AND DISTRIBUTED TEAMS)

Pattern	Enable replication for sites with restricted bandwidth.
Anti-pattern	Use one single "blessed" central repository across all geographical locations.

DON'T force large distributed Enterprise organizations to use a unique central "blessed" repository node, especially when bandwidth is limited and connectivity between sites is unreliable. You may also want to include extra resilience for high availability, even when bandwidth is not an issue.
DO server-side replications across sites by enabling push/pull among "blessed" repositories located in the main geographical locations for development. For example, you may have one or two US-based replicas, plus one for Europe, one for Asia-Pacific, and one for Latin America.

The Bottom Line: Avoid the temptation of the 'good old centralized days', but don't fall in love with peer-to-peer repos blindly

Sometimes companies new to DVCS will go full-blown distributed for no good reason – otherwise you might as well stick with Subversion, right? Wrong. Git is very flexible in terms of distributedness. Just use Git exactly as each team requires.

Unnecessary use of peer-to-peer repositories can also generate significant push-back, especially for large teams that are already unsure about DVCS.

PATTERN#7: PUBLISHED BRANCH STRATEGY

Pattern	Define and publicize your global branch strategy.
Anti-pattern	Allow all developers to define and push to their local branches.

Git lets you define branches and organize them into hierarchical namespaces. For large enterprises, as for large codebases, careful namespace management is essential.

DEFINE your hierarchy of namespaces – start from refs/heads as the base namespace for all branches.

PUBLISH this hierarchy across your organization in multiple formats (e.g. printouts on office walls, commonly-used internal wikis, shared screens, whiteboards) – wherever your developers already store information on strategy

It will then be clear what the branches represent. This clarity will help:

- **Release management**
- **Production support and hot-fix management**
- **Development of sprints and spikes/experiments**

Here's a typical branching scheme in Git:

- **refs/heads/master**
  (main line for future releases)
- **refs/heads/releases/stable-x.y.z**
  (set of branches for integrating changes and stabilizing code for release)
- **refs/heads/user-xyz/mybranch**
  (optional but very useful: namespace for individual users, allowing them to create their own private branches and share them for review)

The Bottom Line: Avoid DVCS anarchy by publishing a branch strategy

DON'T allow small teams to organize branches without any standards.

DO make it an organization-wide decision whether or not to define rules on branching. Git history is not necessarily a hierarchical structure; complete anarchy may lead to obscure branching histories.

PATTERN#8. PER-FEATURE TOPIC BRANCHES

Pattern	Define one topic branch for each individual feature.
Anti-pattern	Tangle feature commits in a single branch history.

Per-feature topic branching offers two advantages:

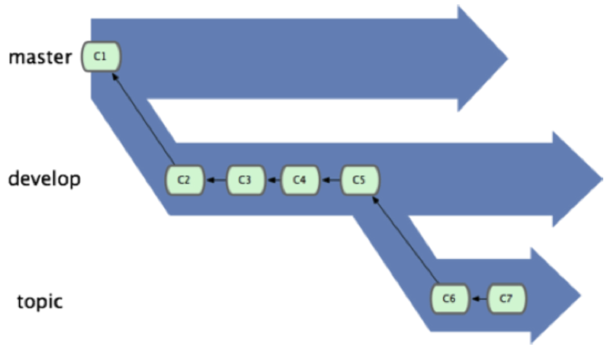
1. It lets features be developed in parallel (which also encourages good coding practices).

2. It lets features be merged based on quality and maturity level per feature (which often don't align across features).

DO define a namespace for all topic branches (e.g. refs/heads/topics/topic-abc)
In a traditional VCS like Subversion, topic branches were typically avoided for two reasons:

1. Branching was easy and cheap but merging was painful and expensive.

2. Consequently, continuous integration of features was feasible only when all topics were in the same branch.



Hot Tip

Advanced Git Servers, such as Google Gerrit Code Review, define additional name-spaces (refs/for/) for managing code-contribution and review, automating the creation and merging of implicit topic-branches (identified by unique "Change-Ids") and validating through scoring the maturity of contributions.

The Bottom Line: Avoid interleaved commits

To get a better gut feel for the problem, consider the following situation.

Feature A has commits A1, A2 and A3; feature B has commits B1, B2 and B3; the master branch has commits A1, B1, A2, B2, A3, B3.

If feature B is de-scoped for any reason (quality, time-to-market, security risks, etc.) the master branch history will have no points that will lead to a stable-xyz release. Then you'll need to cherry-pick A1, A2 and A3 into the main branch, which itself risks conflicts, because of the missing commits in the history that relate to the de-scoped feature.

PATTERN#9: LOCAL-ONLY REBASING

Pattern	Limit rebasing to local repositories or individual branches only.
Anti-pattern	Rebase is powerful so allow its use everywhere, even on remote shared repositories and branches.

git rebase is probably one of the most useful day-to-day commands in Git's hefty arsenal. It works as a time-machine – it lets you repeat a set of changes from a specific point in your project history.

More impressively, commits can be changed, moved or even deleted. You can even create a temporary branch in order to abort the operation and restore the original set of commits. You can imagine how this can be incredibly powerful – when the person in control knows exactly what needs to be done.

Hot Tip

Git keeps unreferenced deleted objects in its store. They can be recovered by looking at Git "reflog" (audit of all references changes). However, when git gc (Garbage Collection) is executed, unreferenced objects are physically removed and cannot be recovered anymore. This is a compliance concern for most enterprises.

But when two developers are working on the same branch - if they both want to rebase the same branch - the last one wins, and the loser isn't even notified that his push has been completely wiped out Therefore:

DON'T push a rebased branch to a remote repository, unless you are absolutely sure that nobody else has made any changes to that branch. If you ever need to force push, always notify everybody before you proceed.

DO consider enterprise-grade management tools that provide history protection. (see pattern #15).

The Bottom Line: Restrict force-push very carefully

When a consistent branching policy has been defined (see Patterns 7 and 8), the problem of preventing history loss is easier to resolve. Developers can simply respect the policies associated with different branch namespaces.

PATTERN#10: BRANCH-LEVEL PERMISSIONS

Pattern	Define one set of permissions for experimental branches, and another for stable release branches.
Anti-pattern	Rely on published policies to enforce branch-level protection.

DO choose a Git server tool that can implement "branch-level" or "fine-grained" permissions (not all currently support this feature)

This will let you open force-push to private branch name-spaces (e.g. refs/heads/user-xyz/mybranch), which will allow developers to benefit from the power of the Git history rewrite.

You should also restrict operations when dealing with common branches such as master development or release.

DO regular code reviews within your development team when pushing to common stable or development branches. Even if you aren't deleting other developers' histories, you could break the build and block team integration tests for hours.

Hot Tip Google Gerrit is the most widely used Git Server that allows both branch-level security and code-review.

The Bottom Line: Hold teams responsible, but use Git permissions to keep master and release branches safe

You might think that only large enterprises need to enforce security at this level, but Git's power can make it dangerous. The difference between a normal and a forced push is just the difference between -f and + on the Git command line. No warning, no confirmation request and a slip of the finger kills an entire branch history.

Git Tools can make mistakes even easier. Once upon a time, a developer innocently clicked "Yes" on the message box "Your push is not fast-forward: do you want to force it?"...

PATTERN#11. GIT PROTOCOL STRATEGY

Pattern	Enforce compatibility of Git protocols with your ICT security standards.
Anti-pattern	Allow Git and SSH protocols just because they are the most widely used and supported.

Part of the genius of Git is the division between the repository logic and protocol logic. You can have a complete Git development and exchange workflow without using any real network protocol implementation. In theory, that makes the network protocol choice more flexible. But in reality, because SSH protocol is so common, this has led to some difficulties.

Workgroups and companies typically have firewalls and ICT Security rules, which are potentially incompatible with some of the protocols available in Git. For example, the Git native protocol does not provide any user authentication layer – so anybody can push and pull from a repository endpoint without an identity credential check.

Other more secure protocols, (e.g. SSH) may be incompatible with your ICT Security requirements – perhaps because of authentication of physical users on a remote Unix system exposed via OpenSSH, or because the authentication keys do not expire and are not archived in the Company LDAP repository. Or more simply: the company firewall might restrict access to port 22. To prevent this from happening:

DO an up-front analysis of the most appropriate protocols for your company and PUBLISH them as standards across your development teams.

DON'T use native Git protocol to push to any central repository. Use it only for peer-to-peer code exchange within your local network.

PATTERN#12. IDENTIFY ENFORCEMENT VIA USER REGISTRY

Pattern	Integrate Git with your existing user registry and existing credentials, and enforce author/committer identity.
Anti-pattern	Allow Git users to commit on behalf of any other internal or external unidentified member.

Git allows peer-to-peer code exchange without any identity or credentials check via a network or security protocol. (Git is designed to allow commits on behalf of a potentially unverified third person.) Because of this, you'll need to enforce identity on your own if you're going to maintain responsibility for commits.

Note that Git distinguishes (a) the Author (who created the original committed code) from (b) the Committer (who amended the commit or pushed the code to a repository). Accordingly:

DO verify the identity of the Git Author according to your company or source code license guidelines. (Often licensing reasons will dictate whether you can allow external team members to contribute code.)

DO verify the identity of the Git Committer (or at least the declared e-mail address) against the user registry used to validate its credentials. For instance, one developer claiming to be linus@torvalds.org (which can be achieved simply by executing "git config user.email linus@torvalds.org") will not be allowed to push unless he proves that he knows the password associated with that e-mail address in the Company User Registry.

The Bottom Line: Avoid accidental flexibility

DON'T allow small workgroups to get rid of security.

Small workgroups may think real identity enforcement is not needed, since the team members know each other and work face-to-face.

But when you don't check identity in the real world, you'll often see many inconsistencies in a Git branch log – missing authors (Git provides a warning but it can be ignored), typos, use/abuse of nick-names. And then it becomes difficult to tell who made what changes to the code.

PATTERN#13: CODE REVIEWS

Pattern	Implement code reviews for distributed teams.
Anti-pattern	Distributed development without code review.

The distributed nature of Git certainly lends itself well to geographically distributed development. But side-by-side peer reviews can be challenging if developers don't share the same office space all the time.

DO codify workflows for code reviews with peers

DO consider additional tools for code review. The leading code review tool is Google Gerrit (free open source)

DO consider automating code build validation by incorporating automation into code reviews (e.g. with Jenkins)

PATTERN#14: HISTORY PROTECTION

Pattern	Enforce security and governance for history protection.
Anti-pattern	Rely simply on ref logs as audit log for Git commits - it's not a true 'audit' log.

Git's 'push –f' command lets you 'rewrite history' by erasing entire code branches and associated ref logs. Avoid the obvious dangers by:

DO factor history protection into your overall Git management strategy

DON'T rely on Git ref logs alone -- they are not true 'audit' logs

DO perform frequent backups of the master repositories

DO consider complementing Git with specialized 3rd party tools to provide Git history protection

See this video for more on history protection for Git: http://www.youtube.com/watch?v=z_FN1NvneBw

PATTERN#15. LIFECYCLE TOOL INTEGRATION

Pattern	Integrate Git into you overall ALM strategy, from a tool and process perspective.
Anti-pattern	Consider Git deployment a stand-alone project , apart from full-scale ALM strategy.

For all its power, Git does not integrate easily with many ALM tools (e.g. trackers, quality management tools, build servers). This will change as Git grows more popular. But then integration quality is likely to suffer, because the Git code base is refactored frequently.

DON'T plan your Git strategy as a silo without considering other tools and processes.
DO discuss and review your Git strategy with project managers, product owners, build managers and quality managers.

DO explore management tools that provide the governance and integration for 3rd party tool integrations while shielding those integration points from a fast-changing Git code base

CONCLUSIONS

Git is a powerful tool for agile development – but the flip-side of flexibility is chaos. Planning and driving Git adoption carefully is necessary to maintain code quality and avoid product delays. The inherent flexibility of DVCS needs to be harnessed at both adoption and iterated development stages.

Further Reading

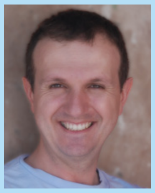
Refcard #94: Getting Started with Git

Various Git resources and downloads

'Go Agile with Git' webinar series

More on Git history protection

ABOUT THE AUTHORS



Luca Milanesio is director and cofounder of GerritForge LLP, a leader in Git and Gerrit for the enterprise and key technology partner of CollabNet Inc. His background includes 20 years of experience in development, software configuration management, and software lifecycle management in large enterprises worldwide. Before GerritForge, Luca was Technical Director and Senior Product of the Security and Compliance platform for Electronic Payments at Primeur in Italy and UK. Luca actively contributes to the Gerrit community and blogs at http://gitenterprise.me and tweets @gitenterprise.

ISBN-13: 978-1-936502-76-9
ISBN-10: 1-936502-76-3

50795

9 781936 502769

$7.95

Version 1.0