Click to verify

Downloadthis free System Design Documenttemplate and use it for your new project. Scroll down to the bottom of the page for the download link. GUIGraphical User InterfaceSDDSystem Design Document Product/SolutionEnvironmentCAVIS II Production Development (Temporary i.e. loan server) Clickhereto download System Design Document Template. However, if you would like to share the information in this article, you may use the link below: important skill for any software engineer is writing technical design docs (TDDs), also referred to as engineering design docs (EDDs). Here in this article I offer some advice for writing good design docs and what mistakes to avoid.One caveat: Different teams will have different standards and conventions for technical design. There is no industry-wide standard for the design process, nor could there be, as different development teams will have different needs depending on their situation. What I will describe is one possible answer, based on my own experience.Design ProcessLets start with the basics: What is a technical design doc, and how does it fit in to the design process?A technical design doc describes a solution to a given technical problem. It is a specification, or design blueprint, for a software program or feature.The primary function of a TDD is to communicate the technical details of the work to be done to members of the team. However, there is a second purpose which is just as important: the process of writing the TDD forces you to organize your thoughts and consider every aspect of the design, ensuring that you havent left anything out.Technical design docs are often part of a larger process which typically has the following steps:Product requirements are defined. These will typically be represented by a Product Requirements Document (PRD). The PRD specifies what the system needs to do, from the perspective of a user or outside agent.Technical requirements are defined. The product requirements are translated into technical requirements what the system needs to accomplish, but now how it does it. The output of this step is a Technical Requirements Document (TRD).Technical design. This contains a technical description of the solution to the requirements outlined in the previous steps. The TDD is the output of this step.Implementation. This is the stage where the solution is actually built.Testing. The system is tested against the PRD and TRD to ensure that it actually fulfills the specified requirements.Between each of these stages there is typically a review process to ensure that no mistakes were made. If any errors, misunderstandings, or ambiguities are detected, these must be corrected before proceeding to the next step.This process is highly variable; the set of steps listed here will change on a case-by-case basis. For example:For smaller features that dont involve a lot of complexity, steps 2 and 3 will often be combined into a single document.If the feature involves a large number of unknowns or some level of research, it may be necessary to construct a proof-of-concept implementation before finalizing the technical design.This process also happens at different scales and levels of granularity. A PRD / TRD / TDD may concern the design of an entire system, or just a single feature. In most environments, the process is also cyclic each design/implement cycle builds on the work of the previous one.The dividing line between TRD and TDD can be a bit blurry at times. For example, suppose you are developing a server that communicates via a RESTful API. If the goal is to conform to an already-established and documented API, then the API specification is part of the requirements and should be referenced in the TRD. If, on the other hand, the goal is to develop a brand new API, then the API specification is part of the design and should be described in the TDD. (However, the requirements document still needs to specify what the API is trying to accomplish.)Writing the TDDThese days, it is common practice to write technical docs in a collaborative document system, such as Google Docs or Confluence; however this is not an absolute requirement. The important thing is that there be a way for your team members to be able to make comments on the document and point out errors and omissions.Most TDDs are between one and ten pages. Although theres no upper limit to the length of a TDD, very large documents will be both difficult to edit and hard for readers to absorb; consider breaking it up into separate documents representing individual steps or phases of the implementation.Diagrams are helpful; there are a number of online tools that you can use to embed illustrations into the document, such as draw.io or Lucidchart. You can also use offline tools such as Inkscape to generate SVG diagrams.The document should be thorough; ideally, it should be possible for someone other than the TDD author to implement the design as written. For example, if the design specifies an implementation of an API, each API endpoint should be documented. If there are subtle design choices, they should be called out.Avoid Common Writing MistakesProbably the most common mistake that I encounter in TDDs is a lack of context. That is, the author wrote down, in as few words as they could manage, how they solved the problem; but they didnt include any information on what the problem was, why it needed to be solved, or what were the consequences of picking that particular solution.Also, its important to keep in mind who the likely reader is, and what level of understanding they have. If you use a term that the reader might not know, dont be afraid to add a definition for it.It hardly needs to be stated that good grammar and spelling are helpful. Also, avoid the temptation for wordplay or cute spelling; while programmers as a class tend to like playing around with language, Ive seen more than one case where excessive frivolity ended up costing the team wasted effort because of misunderstandings. Its all right to use occasional humor or choose colorful, memorable names for features and systems, since that helps people remember them. But dont let your desire to show off how clever you are become a distraction.Speaking of names, choose them carefully; as Mark Twain once wrote, Choose the right word, not its second cousin. Theres a tendency for engineers with poor vocabularies to use the same generic terms over and over again for different things, leading to overloading and confusion. For example, naming a class DataManager is vague and tells you nothing about what it actually does; by the same token a package or directory named utils could contain virtually anything. Consult a thesaurus if you need to find a better word, or better, a specialized synonym database such as WordNet.TDD TemplateWhen writing a TDD, it can be helpful to start with a standard template. The following is a template that I have used in a number of projects. Note that this template should be customized where needed; you are free to delete sections which dont apply, add additional sections, or rename headings as appropriate.Author: IntroductionWhat are you trying to accomplish? Whats wrong with things the way they are now?BackgroundDescribe any historical context that would be needed to understand the document, including legacy considerations.TerminologyIf the document uses any special words or terms, list them here.Non-GoalsIf there are related problems that you have decided not to address with this design, but which someone might conceivably expect you to solve, then list them here.Proposed DesignStart with a brief, high-level description of the solution. The following sections will go into more detail.System ArchitectureIf the design consists of a collaboration between multiple large-scale components, list those components here or better, include a diagram.Data ModelDescribe how the data is stored. This could include a description of the database schema.Interface/API DefinitionsDescribe how the various components talk to each other. For example, if there are REST endpoints, describe the endpoint URL and the format of the data and parameters used.Business LogicIf the design requires any non-trivial algorithms or logic, describe them.Migration StrategyIf the design incurs non-backwards-compatible changes to an existing system, describe the process whereby entities that depend on the system are going to migrate to the new design.ImpactDescribe the potential impacts of the design on overall performance, security, and other aspects of the system.RisksIf there are any risks or unknowns, list them here. Also if there is additional research to be done, mention that as well.AlternativesIf there are other potential solutions which were considered and rejected, list them here, as well as the reason why they were not chosen.Of course, these sections are only starting points. You can add additional sections such as Design Considerations, Abstract, References, Acknowledgements, and so on as appropriate.TDD LifecycleDuring construction of the system, the TDD serves as a reference, coordinating the activities of the team members working on the project. However, after construction is finished, the TDD will continue to exist and serve as documentation for how the system works. You may want to distinguish between current and archived TDDs.However, there are two perils to watch out for:First, TDDs can quickly become out of date as the system continues to evolve. An engineer using a two-year-old TDD as a reference can waste a lot of time trying to understand why the system doesnt behave as described. Ideally, stale TDDs would be marked as obsolete or superseded; in practice this seldom happens, as teams tend to focus on current rather than past work. (Keeping documentation up to date is a challenge that every engineering team struggles with.)Second, a TDD may not include all of the information needed to interface with the system. A TDD might only cover a set of changes to an already-existing system, in which case you would need to consult earlier documentation (if it exists) to get the whole picture. And a TDD mainly focuses on implementation details, which may be irrelevant to someone who simply wants to invoke an API.Thus, a TDD should not be considered an adequate substitute for actual user or API reference docs.FinallyThere are plenty of other articles on the web explaining how to write a great design doc. Dont just read this one! Read several, and then pick a mix of ideas that is right for you.UpdateTheres a follow-on article, Writing Technical Design Documents, Revisited, that provides some additional information.More Essays by Talin:An important skill for any software engineer is writing technical design docs (TDDs), also referred to as engineering design docs (EDDs). Here in this article I offer some advice for writing good design docs and what mistakes to avoid.One caveat: Different teams will have different standards and conventions for technical design. There is no industry-wide standard for the design process, nor could there be, as different development teams will have different needs depending on their situation. What I will describe is one possible answer, based on my own experience.Design ProcessLets start with the basics: What is a technical design doc, and how does it fit in to the design process?A technical design doc describes a solution to a given technical problem. It is a specification, or design blueprint, for a software program or feature.The primary function of a TDD is to communicate the technical details of the work to be done to members of the team. However, there is a second purpose which is just as important: the process of writing the TDD forces you to organize your thoughts and consider every aspect of the design, ensuring that you havent left anything out.Technical design docs are often part of a larger process which typically has the following steps:Product requirements are defined. These will typically be represented by a Product Requirements Document (PRD). The PRD specifies what the system needs to do, from the perspective of a user or outside agent.Technical requirements are defined. The product requirements are translated into technical requirements what the system needs to accomplish, but now how it does it. The output of this step is a Technical Requirements Document (TRD).Technical design. This contains a technical description of the solution to the requirements outlined in the previous steps. The TDD is the output of this step.Implementation. This is the stage where the solution is actually built.Testing. The system is tested against the PRD and TRD to ensure that it actually fulfills the specified requirements.Between each of these stages there is typically a review process to ensure that no mistakes were made. If any errors, misunderstandings, or ambiguities are detected, these must be corrected before proceeding to the next step.This process is highly variable; the set of steps listed here will change on a case-by-case basis. For example:For smaller features that dont involve a lot of complexity, steps 2 and 3 will often be combined into a single document.If the feature involves a large number of unknowns or some level of research, it may be necessary to construct a proof-of-concept implementation before finalizing the technical design.This process also happens at different scales and levels of granularity. A PRD / TRD / TDD may concern the design of an entire system, or just a single feature. In most environments, the process is also cyclic each design/implement cycle builds on the work of the previous one.The dividing line between TRD and TDD can be a bit blurry at times. For example, suppose you are developing a server that communicates via a RESTful API. If the goal is to conform to an already-established and documented API, then the API specification is part of the requirements and should be referenced in the TRD. If, on the other hand, the goal is to develop a brand new API, then the API specification is part of the design and should be described in the TDD. (However, the requirements document still needs to specify what the API is trying to accomplish.)Writing the TDDThese days, it is common practice to write technical docs in a collaborative document system, such as Google Docs or Confluence; however this is not an absolute requirement. The important thing is that there be a way for your team members to be able to make comments on the document and point out errors and omissions.Most TDDs are between one and ten pages. Although theres no upper limit to the length of a TDD, very large documents will be both difficult to edit and hard for readers to absorb; consider breaking it up into separate documents representing individual steps or phases of the implementation.Diagrams are helpful; there are a number of online tools that you can use to embed illustrations into the document, such as draw.io or Lucidchart. You can also use offline tools such as Inkscape to generate SVG diagrams.The document should be thorough; ideally, it should be possible for someone other than the TDD author to implement the design as written. For example, if the design specifies an implementation of an API, each API endpoint should be documented. If there are subtle design choices, they should be called out.Avoid Common Writing MistakesProbably the most common mistake that I encounter in TDDs is a lack of context. That is, the author wrote down, in as few words as they could manage, how they solved the problem; but they didnt include any information on what the problem was, why it needed to be solved, or what were the consequences of picking that particular solution.Also, its important to keep in mind who the likely reader is, and what level of understanding they have. If you use a term that the reader might not know, dont be afraid to add a definition for it.It hardly needs to be stated that good grammar and spelling are helpful. Also, avoid the temptation for wordplay or cute spelling; while programmers as a class tend to like playing around with language, Ive seen more than one case where excessive frivolity ended up costing the team wasted effort because of misunderstandings. Its all right to use occasional humor or choose colorful, memorable names for features and systems, since that helps people remember them. But dont let your desire to show off how clever you are become a distraction.Speaking of names, choose them carefully; as Mark Twain once wrote, Choose the right word, not its second cousin. Theres a tendency for engineers with poor vocabularies to use the same generic terms over and over again for different things, leading to overloading and confusion. For example, naming a class DataManager is vague and tells you nothing about what it actually does; by the same token a package or directory named utils could contain virtually anything. Consult a thesaurus if you need to find a better word, or better, a specialized synonym database such as WordNet.TDD TemplateWhen writing a TDD, it can be helpful to start with a standard template. The following is a template that I have used in a number of projects. Note that this template should be customized where needed; you are free to delete sections which dont apply, add additional sections, or rename headings as appropriate.Author: IntroductionWhat are you trying to accomplish? Whats wrong with things the way they are now?BackgroundDescribe any historical context that would be needed to understand the document, including legacy considerations.TerminologyIf the document uses any special words or terms, list them here.Non-GoalsIf there are related problems that you have decided not to address with this design, but which someone might conceivably expect you to solve, then list them here.Proposed DesignStart with a brief, high-level description of the solution. The following sections will go into more detail.System ArchitectureIf the design consists of a collaboration between multiple large-scale components, list those components here or better, include a diagram.Data ModelDescribe how the data is stored. This could include a description of the database schema.Interface/API DefinitionsDescribe how the various components talk to each other. For example, if there are REST endpoints, describe the endpoint URL and the format of the data and parameters used.Business LogicIf the design requires any non-trivial algorithms or logic, describe them.Migration StrategyIf the design incurs non-backwards-compatible changes to an existing system, describe the process whereby entities that depend on the system are going to migrate to the new design.ImpactDescribe the potential impacts of the design on overall performance, security, and other aspects of the system.RisksIf there are any risks or unknowns, list them here. Also if there is additional research to be done, mention that as well.AlternativesIf there are other potential solutions which were considered and rejected, list them here, as well as the reason why they were not chosen.Of course, these sections are only starting points. You can add additional sections such as Design Considerations, Abstract, References, Acknowledgements, and so on as appropriate.TDD LifecycleDuring construction of the system, the TDD serves as a reference, coordinating the activities of the team members working on the project. However, after construction is finished, the TDD will continue to exist and serve as documentation for how the system works. You may want to distinguish between current and archived TDDs.However, there are two perils to watch out for:First, TDDs can quickly become out of date as the system continues to evolve. An engineer using a two-year-old TDD as a reference can waste a lot of time trying to understand why the system doesnt behave as described. Ideally, stale TDDs would be marked as obsolete or superseded; in practice this seldom happens, as teams tend to focus on current rather than past work. (Keeping documentation up to date is a challenge that every engineering team struggles with.)Second, a TDD may not include all of the information needed to interface with the system. A TDD might only cover a set of changes to an already-existing system, in which case you would need to consult earlier documentation (if it exists) to get the whole picture. And a TDD mainly focuses on implementation details, which may be irrelevant to someone who simply wants to invoke an API.Thus, a TDD should not be considered an adequate substitute for actual user or API reference docs.FinallyThere are plenty of other articles on the web explaining how to write a great design doc. Dont just read this one! Read several, and then pick a mix of ideas that is right for you.UpdateTheres a follow-on article, Writing Technical Design Documents, Revisited, that provides some additional information.More Essays by Talin:As a software engineer, your primary role is to solve technical problems. Your first impulse may be to immediately jump straight into writing code. But that can be a terrible idea if you havent thought through your solution.You can think through difficult technical problems by writing a technical spec. Writing one can be frustrating if you feel like youre not a good writer. You may even think that its an unnecessary chore. But writing a technical spec increases the chances of having a successful project, service, or feature that all stakeholders involved are satisfied with. It decreases the chances of something going horribly wrong during implementation and even after youve launched your product.In this article, Ill walk you through how to write a technical spec that ensures a strong product.A technical specification document outlines how youre going to address a technical problem by designing and building a solution for it. Its sometimes also referred to as a technical design document, or an engineering design document. Its often written by the engineer who will build the solution or be the point person during implementation, but for larger projects, it can be written by technical leads, project leads, or senior engineers. These documents show the engineers team and other stakeholders what the design, work involved, impact, and timeline of a feature, project, program, or service will be.Technical specs have immense benefits to everyone involved in a project: the engineers who write them, the teams that use them, even the projects that are designed off of them. Here are some reasons why you should write one.By writing a technical spec, engineers are forced to examine a problem before going straight into code, where they may overlook some aspect of the solution. When you break down, organize, and time box all the work youll have to do during the implementation, you get a better view of the scope of the solution. Technical specs, because they are a thorough view of the proposed solution, they also serve as documentation for the project, both for the implementation phase and after, to communicate your accomplishments on the project.With this well-thought out solution, your technical spec saves you from repeatedly explaining your design to multiple teammates and stakeholders. But nobodys perfect; your peers and more seasoned engineers may show you new things from them about design, new technologies, engineering practices, alternative solutions, etc. that you may not have come across or thought of before. They may catch exceptional cases of the solution that you may have neglected, reducing your liability. The more eyes you have on your spec, the better.A technical spec is a straightforward and efficient way to communicate project design ideas between a team and other stakeholders. The whole team can collaboratively solve a problem and create a solution. As more teammates and stakeholders contribute to a spec, it makes them more invested in the project and encourages them to take ownership and responsibility for it. With everyone on the same page, it limits complications that may arise from overlapping work. Newer teammates unfamiliar with the project can onboard themselves and contribute to the implementation earlier.Investing in a technical spec ultimately results in a superior product. Since the team is aligned and in agreement on what needs to be done through the spec, big projects can progress faster. A spec is essential in managing complexity and preventing scope and feature creep by setting project limits. It sets priorities thereby making sure that only the most impactful and urgent parts of a project go out first.Post implementation, it helps resolve problems that cropped up within the project, as well as provide insight in retrospectives and postmortems. The best planned specs serve as a great guide for measuring success and return on investment of engineering time.Gather the existing information in the problem domain before getting started. Read over any product/feature requirements that the product team has produced, as well as technical requirements/standards associated with the project. With this knowledge of the problem history, try to state the problem in detail and brainstorm all kinds of solutions you may think might resolve it. Pick the most reasonable solution out of all the options you have come up with.Remember that you arent alone in this task. Ask an experienced engineer whos knowledgeable on the problem to be your sounding board. Invite them to a meeting and explain the problem and the solution you picked. Lay out your ideas and thought process and try to persuade them that your solution is the most appropriate. Gather their feedback and ask them to be a reviewer for your technical spec.Finally, its time to actually write the spec. Block off time in your calendar to write the first draft of the technical spec. Usea collaborative document editor that your whole team has access to. Get a technical spec template (see below) and write a rough draft.There are a wide range of problems being solved by a vast number of companies today. Each organization is distinct and creates its own unique engineering culture. As a result, technical specs may not be standard even within companies, divisions, teams, and even among engineers on the same team. Every solution has different needs and you should tailor your technical spec based on the project. You do not need to include all the sections mentioned below. Select the sections that work for your design and forego the rest.From my experience, there are seven essential parts of a technical spec: front matter, introduction, solutions, further considerations, success evaluation, work, deliberation, and end matter.TitleAuthor(s)TeamReviewer(s)Created onLast updatedEpic, ticket, issue, or task tracker reference linka. Overview, Problem Description, Summary, or AbstractSummary of the problem (from the perspective of the user), the context, suggested solution, and the stakeholders.b. Glossary or TerminologyNew terms you come across as you research your design or terms you may suspect your readers/stakeholders not to know.c. Context or BackgroundReasons why the problem is worth solvingOrigin of the problemHow the problem affects users and company goalsPast efforts made to solve the solution and why they were not effectiveHow the product relates to team goals, OKRsHow the solution fits into the overall product roadmap and strategyHow the solution fits into the technical strategyd. Goals or Product and Technical RequirementsProduct requirements in the form of user storiesTechnical requirementse. Non-Goals or Out of ScopeProduct and technical requirements that will be disregardedf. Future GoalsProduct and technical requirements slated for a future timeg. AssumptionsConditions and resources that need to be present and accessible for the solution to work as described.a. Current or Existing Solution / DesignCurrent solution descriptionPros and cons of the current solutionb. Suggested or Proposed Solution / DesignExternal components that the solution will interact with and that it will alterDependencies of the current solutionPros and cons of the proposed solutionData Model / Schema Changes Schema definitionsNew data modelsModified data modelsData validation methodsBusiness Logic API changesPseudocodeFlowchartsError statesFailure scenariosConditions that lead to errors and failuresLimitationsPresentation Layer User requirementsUX changesUI changesWireframes with descriptionsLinks to UI/UX designers workMobile concernsWeb concernsUI statesError handlingOther questions to answer How will the solution scale?What are the limitations of the solution?How will it recover in the event of a failure?How will it cope with future requirements?c. Test PlanExplanations of how the tests will make sure user requirements are metUnit testsIntegrations testsQAd. Monitoring and Alerting PlanLogging plan and toolsMonitoring plan and toolsMetrics to be used to measure healthHow to ensure observabilityAlerting plan and toolse. Release / Roll-out and Deployment PlanDeployment architectureDeployment environmentsPhased roll-out plan e.g. using feature flagsPlan outlining how to communicate changes to the users, for example, with release notesf. Rollback PlanDetailed and specific liabilitiesPlan to reduce liabilitiesPlan describing how to prevent other components, services, and systems from being affectedg. Alternate Solutions / DesignsShort summary statement for each alternative solutionPros and cons for each alternativeReasons why each solution couldnt workWays in which alternatives were inferior to the proposed solutionMigration plan to next best alternative in case the proposed solution falls througha. Impact on other teamsHow will this increase the work of other people?b. Third-party services and platforms considerationsIs it really worth it compared to building the service in-house?What are some of the security and privacy concerns associated with the services/platforms?How much will it cost?How will it scale?What possible future issues are anticipated?c. Cost analysisWhat is the cost to run the solution per day?What does it cost to roll it out?d. Security considerationsWhat are the potential threats?How will they be mitigated?How will the solution affect the security of other components, services, and systems?e. Privacy considerationsDoes the solution follow local laws and legal policies on data privacy?How does the solution protect users data privacy?What are some of the tradeoffs between personalization and privacy in the solution?f. Regional considerationsWhat is the impact of internationalization and localization on the solution?What are the latency issues?What are the legal concerns?What is the state of service availability?How will data transfer across regions be achieved and what are the concerns here?g. Accessibility considerationsHow accessible is the solution?What tools will you use to evaluate its accessibility?h. Operational considerationsDoes this solution cause adverse aftereffects?How will data be recovered in case of failure?How will the solution recover in case of a failure?How will operational costs be kept low while delivering increased value to the users?i. RisksWhat risks are being undertaken with this solution?Are there risks that once taken cant be walked back?What is the cost-benefit analysis of taking these risks?j. Support considerationsHow will the support team get across information to users about common issues they may face while interacting with the changes?How will we ensure that the users are satisfied with the solution and can interact with it with minimal support?Who is responsible for the maintenance of the solution?How will knowledge transfer be accomplished if the project owner is unavailable?a. ImpactSecurity impactPerformance impactCost impactImpact on other components and servicesb. MetricsList of metrics to captureTools to capture and measure metricsa. Work estimates and timelinesList of specific, measurable, and time-bound tasksResources needed to finish each taskTime estimates for how long each task needs to be completedb. PrioritizationCategorization of tasks by urgency and impactc. MilestonesDated checkpoints when significant chunks of work will have been completedMetrics to indicate the passing of the milestoned. Future workList of tasks that will be completed in the futurea. DiscussionElements of the solution that members of the team do not agree on and need to be debated further to reach a consensus.b. Open QuestionsQuestions about things you do not know the answers to or are unsure that you pose to the team and stakeholders for their input. These may include aspects of the problem you dont know how to resolve yet.a. Related WorkAny work external to the proposed solution that is similar to it in some way and is worked on by different teams. Its important to know this to enable knowledge sharing between such teams when faced with related problems.b. ReferencesLinks to documents and resources that you used when coming up with your design and wish to credit.c. AcknowledgmentsCredit people who have contributed to the design that you wish to recognize.Now that you have a spec written, its time to refine it. Go through your draft as if you were an independent reviewer. Ask yourself what parts of the design are unclear and you are uncertain about. Modify your draft to include these issues. Review the draft a second time as if you were tasked to implement the design just based on the technical spec alone. Make sure the spec is a clear enough implementation guideline that the team can work on if you are unavailable. If you have doubts about the solution and would like to test it out just to make sure it works, create a simple prototype to prove your concept.When youve thoroughly reviewed it, send the draft out to your team and the stakeholders. Address all comments, questions, and suggestions as soon as possible. Set deadlines to do this for every issue. Schedule meetings to talk through issues that the team is divided on or is having unusually lengthy discussions about on the document. If the team fails to agree on an issue even after having in-person meetings to hash them out, make the final call on it as the buck stops with you. Request engineers on different teams to review your spec so you can get an outsiders perspective which will enhance how it comes across to stakeholders not part of the team. Update the document with any changes in the design, schedule, work estimates, scope, etc. even during implementation.Writing test specs can be an impactful way to guarantee that your project will be successful. A little planning and a little forethought can make the actual implementation of a project a whole lot easier.A curated library of our favorite 1000+ design doc examples and templates from 40+ leading engineering organizations and open source projectsAn important skill for any software engineer is writing technical design docs (TDDs), also referred to as engineering design docs (EDDs). Here in this article I offer some advice for writing good design docs and what mistakes to avoid.One caveat: Different teams will have different standards and conventions for technical design. There is no industry-wide standard for the design process, nor could there be, as different development teams will have different needs depending on their situation. What I will describe is one possible answer, based on my own experience.Design ProcessLets start with the basics: What is a technical design doc, and how does it fit in to the design process?A technical design doc describes a solution to a given technical problem. It is a specification, or design blueprint, for a software program or feature.The primary function of a TDD is to communicate the technical details of the work to be done to members of the team. However, there is a second purpose which is just as important: the process of writing the TDD forces you to organize your thoughts and consider every aspect of the design, ensuring that you havent left anything out.Technical design docs are often part of a larger process which typically has the following steps:Product requirements are defined. These will typically be represented by a Product Requirements Document (PRD). The PRD specifies what the system needs to do, from the perspective of a user or outside agent.Technical requirements are defined. The product requirements are translated into technical requirements what the system needs to accomplish, but now how it does it. The output of this step is a Technical Requirements Document (TRD).Technical design. This contains a technical description of the solution to the requirements outlined in the previous steps. The TDD is the output of this step.Implementation. This is the stage where the solution is actually built.Testing. The system is tested against the PRD and TRD to ensure that it actually fulfills the specified requirements.Between each of these stages there is typically a review process to ensure that no mistakes were made. If any errors, misunderstandings, or ambiguities are detected, these must be corrected before proceeding to the next step.This process is highly variable; the set of steps listed here will change on a case-by-case basis. For example:For smaller features that dont involve a lot of complexity, steps 2 and 3 will often be combined into a single document.If the feature involves a large number of unknowns or some level of research, it may be necessary to construct a proof-of-concept implementation before finalizing the technical design.This process also happens at different scales and levels of granularity. A PRD / TRD / TDD may concern the design of an entire system, or just a single feature. In most environments, the process is also cyclic each design/implement cycle builds on the work of the previous one.The dividing line between TRD and TDD can be a bit blurry at times. For example, suppose you are developing a server that communicates via a RESTful API. If the goal is to conform to an already-established and documented API, then the API specification is part of the requirements and should be referenced in the TRD. If, on the other hand, the goal is to develop a brand new API, then the API specification is part of the design and should be described in the TDD. (However, the requirements document still needs to specify what the API is trying to accomplish.)Writing the TDDThese days, it is common practice to write technical docs in a collaborative document system, such as Google Docs or Confluence; however this is not an absolute requirement. The important thing is that there be a way for your team members to be able to make comments on the document and point out errors and omissions.Most TDDs are between one and ten pages. Although theres no upper limit to the length of a TDD, very large documents will be both difficult to edit and hard for readers to absorb; consider breaking it up into separate documents representing individual steps or phases of the implementation.Diagrams are helpful; there are a number of online tools that you can use to embed illustrations into the document, such as draw.io or Lucidchart. You can also use offline tools such as Inkscape to generate SVG diagrams.The document should be thorough; ideally, it should be possible for someone other than the TDD author to implement the design as written. For example, if the design specifies an implementation of an API, each API endpoint should be documented. If there are subtle design choices, they should be called out.Avoid Common Writing MistakesProbably the most common mistake that I encounter in TDDs is a lack of context. That is, the author wrote down, in as few words as they could manage, how they solved the problem; but they didnt include any information on what the problem was, why it needed to be solved, or what were the consequences of picking that particular solution.Also, its important to keep in mind who the likely reader is, and what level of understanding they have. If you use a term that the reader might not know, dont be afraid to add a definition for it.It hardly needs to be stated that good grammar and spelling are helpful. Also, avoid the temptation for wordplay or cute spelling; while programmers as a class tend to like playing around with language, Ive seen more than one case where excessive frivolity ended up costing the team wasted effort because of misunderstandings. Its all right to use occasional humor or choose colorful, memorable names for features and systems, since that helps people remember them. But dont let your desire to show off how clever you are become a distraction.Speaking of names, choose them carefully; as Mark Twain once wrote, Choose the right word, not its second cousin. Theres a tendency for engineers with poor vocabularies to use the same generic terms over and over again for different things, leading to overloading and confusion. For example, naming a class DataManager is vague and tells you nothing about what it actually does; by the same token a package or directory named utils could contain virtually anything. Consult a thesaurus if you need to find a better word, or better, a specialized synonym database such as WordNet.TDD TemplateWhen writing a TDD, it can be helpful to start with a standard template. The following is a template that I have used in a number of projects. Note that this template should be customized where needed; you are free to delete sections which dont apply, add additional sections, or rename headings as appropriate.Author: IntroductionWhat are you trying to accomplish? Whats wrong with things the way they are now?BackgroundDescribe any historical context that would be needed to understand the document, including legacy considerations.TerminologyIf the document uses any special words or terms, list them here.Non-GoalsIf there are related problems that you have decided not to address with this design, but which someone might conceivably expect you to solve, then list them here.Proposed DesignStart with a brief, high-level description of the solution. The following sections will go into more detail.System ArchitectureIf the design consists of a collaboration between multiple large-scale components, list those components here or better, include a diagram.Data ModelDescribe how the data is stored. This could include a description of the database schema.Interface/API DefinitionsDescribe how the various components talk to each other. For example, if there are REST endpoints, describe the endpoint URL and the format of the data and parameters used.Business LogicIf the design requires any non-trivial algorithms or logic, describe them.Migration StrategyIf the design incurs non-backwards-compatible changes to an existing system, describe the process whereby entities that depend on the system are going to migrate to the new design.ImpactDescribe the potential impacts of the design on overall performance, security, and other aspects of the system.RisksIf there are any risks or unknowns, list them here. Also if there is additional research to be done, mention that as well.AlternativesIf there are other potential solutions which were considered and rejected, list them here, as well as the reason why they were not chosen.Of course, these sections are only starting points. You can add additional sections such as Design Considerations, Abstract, References, Acknowledgements, and so on as appropriate.TDD LifecycleDuring construction of the system, the TDD serves as a reference, coordinating the activities of the team members working on the project. However, after construction is finished, the TDD will continue to exist and serve as documentation for how the system works. You may want to distinguish between current and archived TDDs.However, there are two perils to watch out for:First, TDDs can quickly become out of date as the system continues to evolve. An engineer using a two-year-old TDD as a reference can waste a lot of time trying to understand why the system doesnt behave as described. Ideally, stale TDDs would be marked as obsolete or superseded; in practice this seldom happens, as teams tend to focus on current rather than past work. (Keeping documentation up to date is a challenge that every engineering team struggles with.)Second, a TDD may not include all of the information needed to interface with the system. A TDD might only cover a set of changes to an already-existing system, in which case you would need to consult earlier documentation (if it exists) to get the whole picture. And a TDD mainly focuses on implementation details, which may be irrelevant to someone who simply wants to invoke an API.Thus, a TDD should not be considered an adequate substitute for actual user or API reference docs.FinallyThere are plenty of other articles on the web explaining how to write a great design doc. Dont just read this one! Read several, and then pick a mix of ideas that is right for you.UpdateTheres a follow-on article, Writing Technical Design Documents, Revisited, that provides some additional information.More Essays by Talin:

**Software engineering design document template. Design doc software. Engineering design document. Design document template example. Design document template. Engineering template.**